INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

INFORMAČNÍ SYSTÉMY 2

JAROSLAV PROCHÁZKA
MAREK VAJGL
JAROSLAV ŽÁČEK

ČÍSLO OPERAČNÍHO PROGRAMU: CZ.1.07
NÁZEV OPERAČNÍHO PROGRAMU:
OP VZDĚLÁVÁNÍ PRO KONKURENCESCHOPNOST

TVORBA DISTANČNÍCH VZDĚLÁVACÍCH MODULŮ PRO CELOŽIVOTNÍ VZDĚLÁVÁNÍ DLE § 60 ZÁKONA Č. 111/1998 SB. O VŠ NA PŘÍRODOVĚDECKÉ FAKULTĚ OSTRAVSKÉ UNIVERZITY

REGISTRAČNÍ ČÍSLO PROJEKTU: CZ.1.07/3.2.07/02.0033

OSTRAVA 2012

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky

Recenzenti:

Ing. Roman Šmiřák – odborný oponent
RNDr. Martin Kotyrba – metodický oponent

Název: Informační systémy 2
Autor: Jaroslav Procházka, Marek Vajgl, Jaroslav Žáček
Vydání: první, 2012
Počet stran: 197

Studijní materiály jsou určeny pro distanční kurz: Informační systémy 2

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídají autoři.

Obsah

1 Průvodce modulem Informační systémy 2 5

1.1 K čemu slouží Průvodce modulem 5
1.2 Dvě stránky modulu – FORMA a OBSAH 5
1.3 Formální charakteristika modulu Informační systémy 2 6
1.4 Obsahová charakteristika modulu Informační systémy 2 8

2 Agilní přístupy 10

2.1 Scrum 13
2.2 Extrémní programování 17
2.3 Další agilní přístupy 20
2.4 Mylné představy a názory o agile 23
2.5 Možné problémy – na co si dát pozor 25
2.6 Agilní praktiky a techniky 27

3 Lean development 47

3.1 Lean principy, praktiky a nástroje 48

4 Provoz podpora a údržba 52

4.1 Špatný scénář 56
4.2 Lepší scénář podle ITIL 59

5 Správa IT služeb 63

5.1 ITIL® 66
5.2 Stručný přehled procesů ITIL® 69
5.3 Standard IEEE 1219 74
5.4 CobiT® 75
5.5 Shrnutí přístupů k provozu a údržbě IT 76

6 Microsoft .NET Framework 78

6.1 Představení .NET Frameworku 78
6.2 Architektura 81
6.3 Jazyky 83
6.4 Vývojové prostředí 83
6.5 Základní práce v C# 84
6.6 Technologie použitelné v rámci .NET 86

7 Java 89

7.1 Java Card 91
7.2 Java ME 92
7.3 Java SE 96
7.4 Java Enterprise Edition 100
7.5 Servlety 101
7.6 Prezentační logika (JSP, JSTL) 103
7.7 Enterprise JavaBeans (EJB) 105
7.8 Nástroje pro urychlení vývoje 108
7.9 Nástroje pro podporu testování 109
7.10 Java a frameworky 110

8 Webové služby, SOA 117

8.1 Webové služby 117
8.2 SOA 122

9 AJAX 125

9.1 Popis technologie AJAX 125

10 Pokročilý návrh v UML 131

10.1 Diagramy 131
10.2 Stereotypy 132
10.3 Identifikace objektů 132
10.4 Obecné poznámky 133
10.5 Diagram případů užití – Use case diagram 134
10.6 Diagram aktivit – Activity diagram 137
10.7 Diagram tříd – Class diagram 139
10.8 Sekvenční diagram 143
10.9 Diagram komponent 150
10.10 Shrnutí 151

11 Vzory, anti-vzory 153

11.1 Návrhové vzory 153
11.2 Vzory architektur 155
11.3 Procesní vzory a anti-vzory 157
11.4 Anti-vzory 159
11.5 Prozřetelný pan Brooks 166

12 Měkké aspekty vývoje 169

12.1 Týmové role - Belbin 171
12.2 Typologie osobnosti MBTI 172
12.3 Tým 179
12.4 Koučing a mentoring 181

13 Aplikace informačních systémů – Business Inteligence 186

13.1 Základní principy řešení BI 188
13.2 Aplikace BI 192

14 Literatura 194

1 Průvodce modulem Informační systémy 2

Modul je realizován v rámci projektu ESF VK **Tvorba distančních vzdělávacích modulů pro celoživotní vzdělávání dle §60 zákona č.111/1998 Sb. o VŠ na PřF OU**
Registrační číslo projektu: CZ.1.07/3.2.07/02.0033

1.1 K čemu slouží Průvodce modulem

Cílem Průvodce modulem je seznámit zájemce o studium s obsahovým zaměření a optimálním způsobem studia modulu distanční formou, dále poskytnout informace o tutoriálech, korespondenčních úkolech, seminární práci a podmínkách ukončení modulu, konkrétně bodové limity pro zápočet, zkoušku a výsledné hodnocení.	Cíl Průvodce kurzem

1.2 Dvě stránky modulu – FORMA a OBSAH

Forma modulu je distanční, což představuje především samostudium s využitím PC připojeného k internetu, SW výukového řídicího systému Moodle. Tento systém slouží ke komunikaci s garantem modulu a lektory modulu (konzultace ke studované látce, plnění korespondenčních úkolů, semestrálních projektů aj.) a se spolužáky studujícími stejný modul. Presenční forma studia probíhá formou tutoriálů, které jsou věnovány organizačně technickým stránkám samostudia, a také ke konzultacím k dané problematice. Prezenční forma je doplněna vypsanými konzultacemi, přesné termíny jsou vždy aktualizovány v LMS Moodle, aby účastníci kurzu s nimi byli seznámeni. Závěrečná zkouška/zápočet je vždy realizována prezenční formou na fakultě.	Forma modulu
Obsahově je modul zaměřen hlavně na rozšíření znalostí v oblasti metodik vývoje, provozu a údržby SW/IS a také na technologie, modely a architektury používané při vývoji moderních komplexních IS, jedná se o SOA, RIA či cloud model a jejich použití v technologiích Java a .NET. Obsahem jsou také vzory a anti-vzory jelikož jsou v technologické oblasti důležitým způsobem sdílení znalostí a úspěšných praktik. Nezbytnou součástí vývoje a provozu IS je týmová práce a komunikace, proto se zaměříme také na tzv. měkké dovednosti, týmovou práci a typologii MBTI.	Obsah modulu

1.3 Formální charakteristika modulu Informační systémy 2

1.3.1 Rozsah modulu

• **Úvodní tutoriál** Seznámení se způsobem komunikace s garantem a lektorem modulu, způsobem komunikace s ostatními studenty. Student bude rovněž seznámen s obsahem vybraného modulu a se všemi požadavky na jeho úspěšné zvládnutí. Dále bude účastník modulu seznámen s termíny odesílání korespondenčních úkolů a jejich hodnocením. • **Samostudium** je důležitou součástí studia modulu. Jde o získávání znalostí a dovedností v oblasti vývoje komplexních IS podle moderních trendů (zvláště pochopení principů, smyslu fází a konceptu iterací), zpracování a odesílaní korespondenčních úkolů. • **Další tutoriály** V rámci tutoriálů budou řešeny vyslovené problémy se zvládnutím stěžejních oblastí studia modulu, se složitějšími kapitolami modulu. Tutoriály jsou převážně řešeny jako konzultační, z tohoto důvodu musí být účastníci modulu připraveni na dotazy, které se budou týkat zaslaných korespondenčních úkolů. • **Závěrečný tutoriál** Tento tutoriál bude věnován diskuzi k odevzdaným semestrálním projektům. • **Zkouška/zápočet** Zkouška/zápočet vždy probíhá prezenční formou.	Časový harmonogram

1.3.2 Komunikace v modulu

Komunikace mezi studentem a tutorem (lektorem), mezi studenty probíhá v prostředí Moodle. Ve výjimečných případech lze použít e-mail nebo telefon. Systém LMS Moodle slouží také k odevzdávání korespondenčních úkolů, semestrálních prací, aj.	Způsoby komunikace

1.3.3 Charakteristika účastníků – cílová skupina projektu

Cílovou skupinou jsou zájemci o studium v distančních formách výuky akreditovaných studijních programů na PřF OU nebo na dalších univerzitách s příbuznými obory.	Cílová skupina

Zájemci dostanou příležitost začít studium právě s využitím § 60 zákona č.111/1998 Sb., a tím se na další studium v akreditovaných programu připravit absolvováním nabízených modulů, které jim mohou být uznány v případě přijetí ke studiu akreditovaných oborů. Předpokladem přijetí ke studiu nabízených modulů je středoškolské vzdělání ukončené maturitní zkouškou. Cílovou skupinou jsou například zaměstnanci počítačových i jiných firem bez VŠ vzdělání, případně se vzděláním v jiných oborech. Dále absolventi středních škol, kteří vstoupili na trh práce, ale uvažují o dalším vzdělávání distanční formou, zájemci o informační technologie. Moduly projektu jsou určeny i těm zájemců, kteří jsou vedeni na úřadě práce a snaží se získat zaměstnání získáním nových znalostí, absolvováním dalších rekvalifikací apod.	
Absolvent modulu získá znalosti o Agilních metodikách, moderních architekturách, technologiích a modelech tvorby, provozu a údržby SW/IS a také o měkkých aspektech, týmové práci a typologii osobnosti.	Popis absolventa modulu

1.3.4 Kritéria hodnocení a způsoby prověřování znalostí v modulu

Vedoucí modulu/kurzu hodnotí splnění základních parametrů úkolů s dvěma stupni splnění: splnil – nesplnil.	Jak a kdo hodnotí samostatné práce
Vypracování zadaných korespondenčních úkolů	Požadavky na zápočet/zkoušku
Diskuse nad vypracovanými korespondenčními úkoly včetně ověření teoretické znalosti	Jak probíhá zápočet/zkouška

1.3.5 Doplňující informace

Student bude ke studiu potřebovat PC s připojením na Internet. Dále doporučuji si zajistit doporučenou literaturu (alespoň k nahlédnutí v době přípravy na zkoušku). Průběžně si student bude muset nainstalovat na svůj počítač příslušný software, který je uveden v modulu (case nástroj pro modelování UML diagramů, IDE pro možnou tvorbu a zkoušku fragmentů kódu), jelikož je nutný k vypracování korespondenčních úkolů.	

1.4 Obsahová charakteristika modulu Informační systémy 2

1.4.1 Anotace modulu

Náplní předmětu je pohled na jiné než rigorózní metodiky vývoje a rozdíly oproti jejich postupu a technikám a pokrytí celého životního cyklu IS zahrnující také provoz a údržbu podle standardů ITSM (ITIL). Dalším zaměřením předmětu je představení moderních modelů architektur (cloud, SOA, RIA) a nejvíce používaných technologií jako je .NET a Java a bližší vysvětlení použití jazyka UML. Součástí náplně jsou také různé varianty vzorů (návrhové, procesní, architektur) a také anti-vzory. Moderní vývoj software je díky své komplexnosti také o komunikaci a spolupráci, proto se zaměříme také na tuto oblast.	Anotace

1.4.2 Jaké jsou požadavky na předchozí znalosti a vybavení studujících

Tento předmět rozšiřuje znalosti nabyté v modulu Informační systémy 1 a opět také integruje dosud nabyté znalosti studentů v ostatních předmětech, resp. znalosti z rozličných oborů. Nutným předpokladem je znalost problematiky Informačních systémů 1 a také opět základní znalost architektur počítačů a hardware, sítí, programování, aplikací software a databází. Nezbytné je pak objektově orientované myšlení a znalost objektově orientovaných principů jako základ objektově orientované analýzy a návrhu.	Požadavky na předchozí znalosti

1.4.3 Podrobná osnova modulu

Zaměříme se na následující témata: 1. Agilní metodiky - agile manifesto, zásady, postupy, rozdíly oproti rigorózním metodikám (XP, Scrum, FDD). 2. Lean Software Development - principy, trendy, techniky. 3. Provoz a údržba IS (principy a metody). 4. Správa IT služeb (koncept IT služby, SLA, ITIL). 5. Vlastnosti moderních programovacích jazyků (.NET, Java, Ruby, …). 6. SOA a Cloud modely (podnikové architektury) 7. Web Services (pojmy, XML, UDDI, SOAP, WSDL), AJAX (tvorba interaktivních webových aplikací (koncept, architektura, technologie). 8. Principy syntaxe UML a pokročilé UML modely. 9. Procesní, návrhové vzory, vzory architektur. 10. Antivzory, jejich význam a struktura (vývojářské, manažerské, architektur). 11. Komunikace a týmová práce při vývoji SW (principy RUP, tvorba týmu).	Osnova

12. Měkké dovednosti při vývoji SW (typologie MBTI).	

1.4.4 Klíčová slova modulu

Agilní metodiky vývoje SW/IS, provoz a údržba, ITIL, .NET, Java, AJAX, SOA, Cloud, UML, vzory, anti-vzory, komunikace, soft skills, MBTI.	Klíčová slova

1.4.5 Doplňující literatura

PROCHAZKA, J., KLIMES, C. *Provozujte IT jinak. Agilní a štíhlý provoz, podpora a údržba informačních systémů a IT služeb.*. Praha Grada, 2011. Kadlec, V. *Agilní programování. Computer press, 2004*. Brown, W. J. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998. Šešera, Mičovský, Červeň. *Datové modelování v příkladech. BEN. 2002.*. Gamma E., Helm R., Johnson R., Vlissides J. *Návrh programů pomocí vzorů. Grada 2003*. Buchalcevová, A. *Metodiky vývoje a údržby informačních systémů. Grada. 2005.*. Paleta, P. *Co programátory ve škole neučí aneb softwarové inženýrství v reálné praxi. Computer press. 2003.*	Doplňující informace k modulu

2 Agilní přístupy

V této kapitole se dozvíte:

- Co jsou to agilní přístupy?
- Co je to agilní manifest?
- Jaké jsou principy a techniky agilních přístupů?

Po jejím prostudování byste měli být schopni:

- Pochopit rozdíly mezi klasickým a agilním vývojem.
- Popsat základní praktiky jako je párová práce či TDD.
- Porozumět všem těm pojmům a zkratkám.

Klíčová slova této kapitoly:

Agilní přístupy, agilní manifest, párové programování, refaktoring, neustálá integrace.

Doba potřebná ke studiu: 6 hodin

Průvodce studiem

Kapitola představuje agilní přístupy, jejich principy a také důvody vzniku těchto přístupů. Jednou z částí je také představení praktik a technik, které jsou propagovány hlavně agilními přístupy jako je párové programování, testy řízený vývoj (TDD) či neustálá integrace.
Na studium této části si vyhraďte 6 hodin.

V posledních přibližně deseti letech se rychle rozšířili tzv. agilní přístupy (jmenovitě Scrum, XP či Feature driven development) k vývoji software, stávají se alternativou k vodopádovému způsobu vývoje. Budeme je nazývat přístupy spíše než metodiky, jelikož se nejedná o metodiky (nedefinují přesný popis aktivit, jejich sled, role, nástroje), ale spíše o procesní frameworky (z nichž si vytvoříme svůj vlastní proces podle potřeb projektu a zkušeností členů týmu) nebo o sady principů (jejichž naplněním vyvíjíme agilně). Proč se tyto přístupy nazývají agilní? Toto slovo bylo zvoleno záměrně, jelikož v 90. letech minulého století byl velmi moderním pojmem rychlý vývoj aplikací (angl. *rapid*). Z této módní vlny také pochází nástroje jako RAD (*Rapid Application Development*), které však nebyly moc úspěšné. Jedním z hlavních problémů při vývoji SW totiž není rychlost, ale schopnost reakce na změnu a problémy, které jsou běžné a přichází ze spousty oblastí (trh, zákazník, technologický vývoj, módní vlny, zákony apod.). Agilní tedy neznamená rychlý, ale spíše svižný, živý, flexibilní, přizpůsobivý, což je podstatný rozdíl.

Pokud mluvíme o agilních přístupech, musíme hned na úvod zmínit manifest agilního vývoje, jelikož je, resp. principy v něm zmíněné, jakýmsi společným základem pro všechny přístupy. Tento manifest vznikl v roce 2001 v Utahu v USA, kde se sešli profesionálové z IT oboru (jen namátkou jmenujme Kenta Becka, Alistaira Cockburna či Martina Fowlera), kteří již pracovali na alternativních přístupech k těm tradičním. Všichni za sebou měli zkušenosti z různých, tradičně vedených projektů, jak těch úspěšných, tak těch neúspěšných a jejich cílem byla změna, snaha přinést lepší postupy, které by

uspokojovaly potřeby zákazníků a jejich podnikání a lépe reagovaly na změny. Výsledkem byl agilní manifest, který deklaruje určité hodnoty společné pro všechny tyto přístupy.

Obr. 2-1: Agilní manifest a jeho základní 4 hodnoty; dole původní signatáři (zdroj: www.agilemanifesto.org).

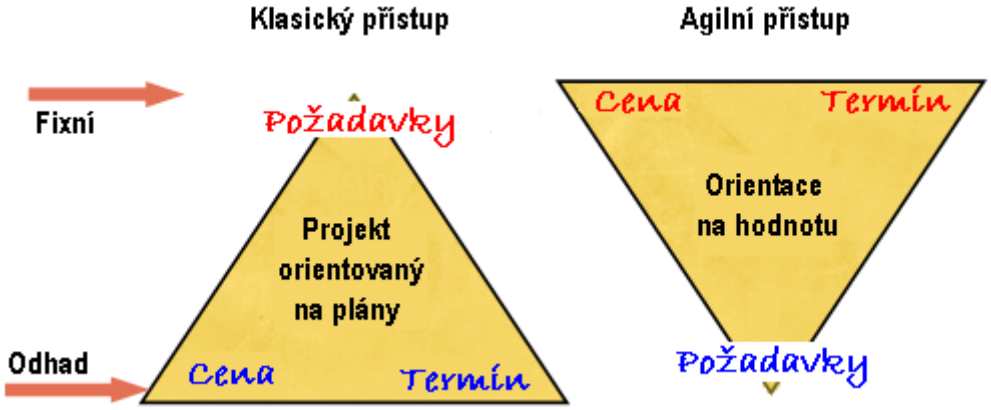
Kromě čtyř hodnot znázorněných na obrázku manifestu, definuje manifest také základní principy, několik volně přeložených si také představíme:

- Vyhovět zákazníkovi, splnit jeho přání.
- Měnící se požadavky jsou vítány (je to konkurenční výhoda pro zákazníka, podpoří to jeho podnikání).
- Časté dodávky fungujícího software zákazníkovi.
- Každodenní spolupráce vývojářů a lidí z businessu z důvodu pochopení jejich potřeb.
- Osobní komunikace tváří v tvář, která je efektivnější a zamezuje chybné interpretaci, která běžně nastává komunikujeme-li na základě dokumentů.

- Základním a hlavním měřítkem postupu je fungující software, ne dokumenty, modely a specifikace, jelikož software přináší zákazníkovi přidanou hodnotu.
- A další...

Pokud v našem procesu vývoje software následujeme tyto principy či naplňujeme pravidla, dostaneme iterativní proces, který produkuje na konci každé iterace inkrement, jímž je kompletně implementovaná nějaká fungující část aplikace. Již v Informačních systémech 1 jsme se zmínili o tom, že RUP (*Rational Unified Process*) a OpenUP jsou také agilní přístupy, pokud jim rozumíme a následujeme jejich principů. Základem RUP/OpenUP je, že náš proces je řízen riziky (nejdříve se snažíme vyřešit možné problémy), zaměřen na architekturu (stabilní základ a základní use casy pro další práce – pozor, ale ne práce do šuplíku jako v případě vodopádu), řízen use casy (pomocí nich plánujeme iterace, jejich scénáře průběžně implementujeme a popisy znovupoužijeme pro testy a dokumentaci). Agilní přístupy často deklarují jednoduchou, minimalistickou, ještě fungující, ale zároveň dostatečnou architekturu. V průběhu dalších iterací probíhá refaktoring a průběžné zlepšování architektury, což je umožněno díky unit testům, jinak bychom mohli vnést více chyb než užitečných zlepšení. Dále tyto přístupy kladou důraz na automatizaci ve formě automatických testů, buildů a také na refaktoring. Je tedy zřejmé, že kritickým faktorem je podpora nástroji a automatizace. V zásadě tedy platí, že pokud budeme kombinovat agilní přístupy s některými RUP postupy a technikami, dostaneme velmi kvalitní a efektivní proces vývoje.

Základem agilních přístupů je důraz na vlastní software a jeho doručení, ale také na komunikaci, a to nejen mezi členy týmu, ale také mezi vývojovým týmem a zákazníkem, resp. uživateli systému. Dále je v agilním manifestu kladen důraz na neustálé doručování software a na funkční software jako primární metriku postupu na projektu, tedy ne specifikace požadavků či návrhové dokumenty a modely. Posledním důležitým bodem je jednoduchost v průběhu vývoje (předcházení plýtvání – vzpomeňte na koláčový graf v Chaos reportu analytiků Standich ukazující velmi nízké využití implementovaných funkčností) a neustálé učení se v průběhu projektu a aplikování těchto naučených věcí.



Obr. 2-2: Trojúhelník kvality v klasických a agilních projektech.

Předchozí obrázek ukazuje důležitý aspekt agilních přístupů, tím je důležitost práce s požadavky a jejich prioritou (potřebou), tzv. ***scope management***. Klasický trojúhelník kvality říká, že výsledná kvalita je dána jeho třemi vrcholy: (1) cenou projektu, (2) dobou na něj potřebnou a (3) množstvím implementovaných požadavků. Kvalita je pak výsledek těchto proměnných a nachází se uprostřed tohoto trojúhelníku. Je to zřejmé, například kvalitnější High-Availability (vysoká dostupnost) řešení ovlivní vrchol cena, tedy bude logicky dražší. Pokud všechny tři zmíněné atributy projektu nereálně zafixujeme, potom kvalita jako výstup této fixace trpí. Vývojový tým si musí jednu z tří proměnných vybrat. Buď čas nebo cenu nebo objem požadavků. Pokud svážeme všechno, jediná možnost, kde se dá takzvaně uhnout je kvalita. Agilní přístupy tento troúhelník otočily. Jako hlavní je dohodnutá cena za dílo a dodání včas, tým se pak snaží za daných omezení dodat co nejvíce potřebných požadavků.

Bohužel se dnešní trh musí řídit mnohdy nesmyslnými předpisy úředníků, kteří o trojúhelníku kvality nemají ani ponětí a proto třeba EU direktivy požadují přesnou fixaci požadavků a času a cena vyhrává nejnižší. Každý člověk trošku znalý matematiky, resp. geometrie pak hned pozná, že trojúhelník kvality bude zákonitě deformovaný. Výsledkem takového zadání bude nízká kvalita, resp. kvalita nižší než bylo očekáváno.

2.1 Scrum

Scrum je v dnešní době velmi populární framework pro iterativní řízení projektů. Ruku v ruce s jeho popularitou jde i neznalost toho, že Scrum je ***jen framework pro projektové řízení***, nepopisuje a nezmiňuje žádné inženýrské postupy. Je na samo-řízeném týmu, jaké inženýrské techniky si zvolí. Z této definice jasně plyne, že pro nezkušený tým může generovat použití Scrumu spoustu problémů (chybějící zaměření na architekturu, rizika, ani slovo o testování), což se také v praxi děje a proto se týmy mnohdy uchylují zpět k původnímu modelu vývoje.

Pojem *scrum* je anglickým výrazem pro mlýnici v rugby, která se vyznačuje spoluprácí hráčů různých rolí a odpovědností za účelem dosažení cíle – získání míče. Paralela s vývojem software je tedy jasná. Scrum byl představen pány Kenem Schwaberem a Mikem Beedlem v roce 1995 (opět tedy můžeme vidět, že se nejedná o módní vlnu, kdežto o přístup používaný téměř 20 let).

Nejdříve si představíme základní pojmy Scrumu. Iterace ve Scrumu se jmenuje ***sprint*** a jejím cílem je vytvořit potenciálně doručitelný kód do provozního prostředí zákazníka (ve Scrumu zvaný *potentially shippable code*). Dalším důležitým pojmem a základním artefaktem je tzv. ***Product backlog***. Jedná se o seznam požadavků ve formě scénářů, respektive tzv. ***user stories*** požadovaných zadavatelem. Tyto požadavky jsou seřazeny podle priorit od nejdůležitější k nejméně důležité. *Product backlog* slouží jako výchozí bod pro plánování sprintu, kdy tým společně s vlastníkem produktu ze strany byznysu (*Product owner*) vybírá scénáře k implementaci (podle odhadu složitosti a znalosti, kolik bodů je tým schopný v rámci sprintu implementovat).

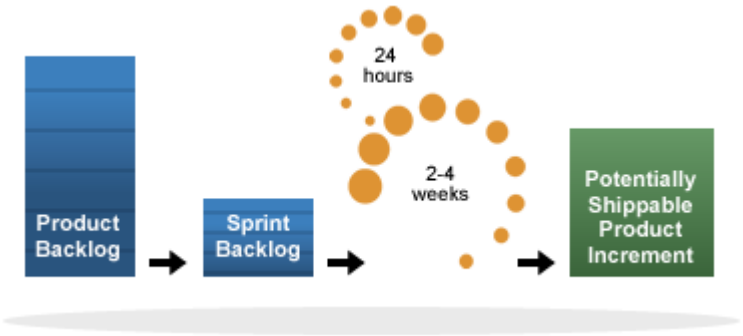
Role ve Scrumu existují pouze tři a veškerá odpovědnost je rozdělena mezi ně:

- ***Product Owner (PO)*** – byznys člověk reprezentující zájmy všech na projektu, resp. výsledném produktu. Vytváří původní požadavky na projekt (*Project backlog*), definuje cíle a základní roadmap projektu. Zařazuje změny do backlogu a pravidelně ho re-prioritizuje.
- ***Scrum Master (SM)*** – role, která primárně pomáhá odstraňovat a řešit problémy a překážky vývojového týmu. Dále je role odpovědná za proces, jeho implementaci, změny a rozšíření znalosti mezi členy týmu. Je odpovědný za to, že každý člen týmu následuje pravidla a praktiky definované Scrumem. Pozor, rozhodně se nejedná o projektového manažera, jak ho známe z tradičních projektů! Jde o pomocníka, který vytváří vhodné prostředí týmu!
- ***Člen týmu, resp. tým*** – odpovědný za vývoj funkcionalit systému, tým je samo-řízený (*self-managed*) a obsahuje všechny potřebné funkce, role. Tým je odpovědný za rozhodnutí, jakým způsobem přetransformovat *product backlog* do formy funkčních inkrementů během jednotlivých sprintů. Celý tým je společně odpovědný za úspěch či neúspěch každého spintu projektu stejně jako celého projektu.

Artefakty Scrumu jsou tedy:

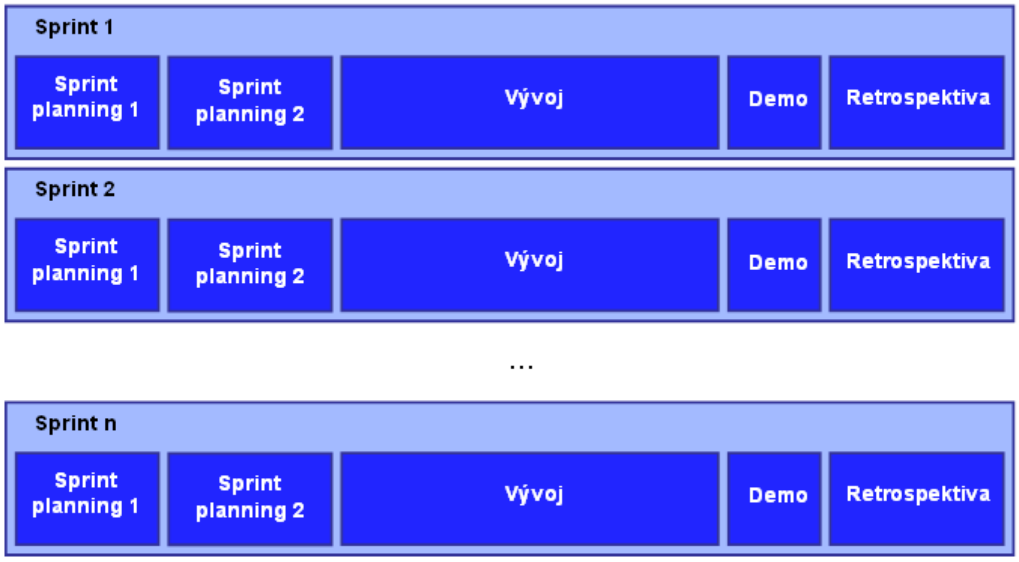
- ***Product backlog*** – seznam požadavků (scénářů) seřazený podle priorit od nejdůležitější až po nejméně důležitou. Obsahuje počáteční odhad kompexnosti, pracnosti a měl by také obsahovat definici, kdy je daný požadavek hotov (pro akceptaci na konci sprintu). *Product backlog* je spravovaný *Product Ownerem*, zapracovává do něj změny a re-prioritizuje. Product backlog slouží jako zdroj požadavků pro sprint.
- ***Sprint backlog*** – definuje vybranou práci a úkoly pro aktuální sprint, které budou na konci sprintu doručeny ve formě inkrementu funkčnosti. Tým definuje úkoly (v rozsahu několika hodin, max. 2 dnů), které je nutné vykonat k doručení vybraných scénářů.
- ***Potenciálně doručitelný produkt*** (*potentially shippable product*) – jeden inkrement funkcionality, jako výstup sprintu, který lze nasadit zákazníkovi do provozu.

Scrum proces vychází z Demingova PDCA (*Plan-Do-Check-Act*) cyklu zlepšování procesů a jeho průběh je následující:



Obr. 2-3: Scrum proces (zdroj: ScrumAlliance.org)

Proces vývoje podle Scrum probíhá zhruba následovně. Tým společně s *Product Ownerem* vybere z *Product backlogu* nejprioritnější scénáře do Sprint backlogu a ty zpracovává v následujícím sprintu. Délka sprintu je typicky od 1-2 týdnů do jednoho měsíce. Na obrázku je doporučeno 2-4 týdny. Jako kontrolní mechanismus, který umožňuje odhalit problémy a překážky, slouží pravidelné denní mítinky (detaily viz dále). Výsledkem sprintu je pak potenciálně doručitelný produkt, který může, ale nemusí být doručen zákazníkovi do provozního prostředí. V úvodu sprintu probíhá plánování se zákazníkem (*Sprint planning 1*), kde vybereme scénáře pro implementaci do následujícího sprintu. Poté následuje další schůzka, kde tým vybrané scénáře rozdělí do úkolů a doplní případnými dalšími identifikovanými úkoly z oblasti architektury či návrhu. Každý z úkolů by měl mít délku trvání přibližně od 4 hodin do maximálně 2 dnů. Na konci sprintu je pak provedena demonstrace zákazníkovi, akceptace dodaného podle evaluačních kritérií (tzv. *definition of done*) a také retrospektiva (popis opět viz níže). Průběh sprintu ukazuje následující obrázek:



Obr. 2-4: Obsah, struktura každého sprintu

Jelikož je Scrum pouze framework pro řízení projektů a tudíž nepopisuje jakým způsobem provádět inženýrské disciplíny (sběr požadavků, analýzu, návrh, implementaci, testování, sestavení), je třeba zmínit možné problémy a pasti Scrumu:

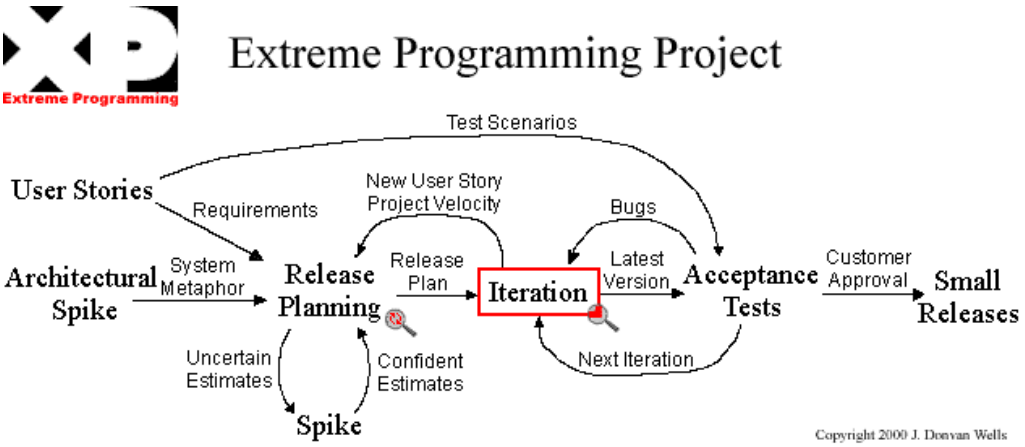
- Scrum explicitně nezmiňuje správu a rizik a práci s nimi. Scrum není řízen riziky, tudíž může nastat problém v pozdějších fázích projektu, když se neidentifikovaná rizika začnou stávat skutečností, jelikož se jim nesnažíme předejít či zmírnit jejich dopad.
- Chybí zaměření na architekturu, což může způsobovat tzv. špagetový kód (více viz kapitola o vzorech), nutnost častého refaktorování a velmi složitou a chybovou údržbu.
- Možné opomenuté závislosti při použití user stories místo use case. Use case spojuje jednotlivé scénáře jako obálka, kdežto user stories jsou často jen fragmenty scénářů. Nyní však již ve Scrum existují tzv. témata a epics, které sdružují příbuzné stories.

- Porušování pravidel Scrumu – pozor na tento častý jev, následování pravidel nám pomůže včas odhalit problémy, pokud je těžké pravidla následovat a odchylujeme se od nich ke starému způsobu práce, znamená to, že existují překážky, na které je nutné se zaměřit a odstranit je (úloha Scrum Mastera a managementu společnosti).

Díky tomu, že je Scrum pouze nástrojem pro projektové řízení, je nutné (a výhodné) ho kombinovat například s XP, RUP, OpenUP či jinou inženýrskou technikou, která týmu řekne, jak se postavit k vývojářským krokům (co, kdy, kdo má dělat na projektu). Pro zkušené senior vývojáře to nemusí být problém, ale pro tým s většinou juniorů je to mnohdy pohroma. Scrum ale díky své jednoduchosti umožnil široké rozšíření povědomí a pokusy o implementaci Agilních principů.

2.2 Extrémní programování

Pravděpodobně nejznámějším agilním přístupem je Extrémní programování (*eXtreme Programming*, ve zkratce XP). Kolem tohoto přístupu panuje řada nepřesností a pověr, stejně jako kolem všech agilních přístupů. Pravěpodobně je to dáno tím, že autor XP, Kent Beck, některé techniky a principy opravdu vyhnal do extrému, proto mohou být některé myšlenky velmi snadno dezinterpretovány. XP definuje psaní kódu jako klíčovou činnost, proto se může zdát, že neděláme analýzu, návrh, natož dokumentaci. V XP ale všechny tyto nezbytné inženýrské činnosti vykonáváme, jen trošku jinak, než jsme zvyklí z tradičních metod.



Obr. 2-5: Životní cyklus projektu podle XP (zdroj: www.extremeprogramming.org)

Dalo by se říct, že XP vlastně nepřináší nic nového, pouze opašuje všeobecně známé inženýrské/programátorské postupy, které byly známy již od dob prvních programátorů. Jakmile ale byly odhaleny slabiny těchto postupů, byly opuštěny a nahrazeny postupy s vyšší režií. XP se snaží oprášit tyto známé postupy, trvá na nich a slabé stránky jednoho postupu se snaží zakrýt silnými jiného postupu.

XP je postaveno na **4 základních hodnotách**, jimiž jsou:

- Komunikace – toto bývá většinou skrytá příčina potíží projektů, nějaký programátor neřekne jinému o kritické změně v návrhu; jindy

programátor nepoloží zákazníkovi tu správnou otázku, takže nedojde k rozhodnutí v byznys oblasti; nebo manažer nepoloží správný dotaz programátorovi a dojde ke zkreslení pokroku na projektu. Vlastností XP je, že řadu postupů nelze bez komunikace provádět (párové programování, plánování a odhadování atd.). XP také využívá kouče/mentora, který zajišťuje opětovnou komunikaci mezi lidmi, pokud nastane její výpadek.

- Jednoduchost – XP sází na jednoduchost; ptáme se, co je nejjednodušší věc, která by mohla fungovat, nehledíme do budoucnosti, že budeme určitě potřebovat ještě to či ono. XP hraje loterii, sází na to, že je lepší udělat jednoduchou věc dnes a zaplatit o něco víc zítra za případnou změnu, než udělat složitější věc dnes, která se ale nemusí vůbec využít.
- Zpětná vazba – konkrétní zpětná vazba o systému je neocenitelná věc, pokud chceme něco vědět, napišme na to test, test nám dá přímou odpověď. Zpětná vazba se týká také zákazníků – vidíme, zda píší lepší, přesnější zadání v průběhu dalších iterací. Pro programátory je také důležitá zpětná vazba od zákazníka. Je naimplementovaná funkce doopravdy to, co potřeboval, to co definoval v zadání? Je spokojený s kvalitou řešení? Je ovládání vhodné?
- Odvaha – poslední hodnotou je pak odvaha. Proto, aby fungovali tyto hodnoty, je zapotřebí mít odvahu. Pokud odhalíme pozdě v projektu chybu architektury, musíme mít odvahu ji vyřešit i za cenu rozbití většiny unit testů a nutnosti jejich předělání. Další odvážnou věcí je zahození již napsaného zdrojového kódu. Někdy řešíme nějaký problém celý den bez valného výsledku, ráno přijdeme a řešení máme za pár desítek minut. Někdy se můžeme pokusit rozvést více návrhů architektury, abychom viděli, jak se s nimi pracuje. Potom zdrojový text zahodíme a začneme psát znovu zdrojový text pro tu nejslibnější architekturu. Odvaha zahodit zdrojový kód či dokonce kód testů je důležitá.

Základní hodnoty, na kterých se společnost či tým shodne, nám pomáhají k následování společných cílů, ne jedinečných krátkodobých zájmů každého jedince. V případě softwarového projektu je toto zřejmé, chceme společně vyprodukovat kvalitní software, který přináší hodnotu zákazníkovi. Naším cílem není krátkodobá zábava jednotlivých členů týmu. Proto XP definuje 4 základní hodnoty, které musí každý člen XP týmu vzít za své, aby společně dosahovali definovaných cílů. Z tohoto faktu vyplývá, že XP není pro každého. Jedinci, kteří neradi spolupracují a egoisticky prosazují pouze svoje řešení, nemají v XP místo. To ale platí pro veškeré agilní přístupy, jelikož jsou založeny na vzájemném respektu a komunikaci.

Základní postupy softwarového vývoje, které XP oprašuje, jsou následující:

- Plánovací hra – rychle určíme rozsah zadání (*scope*) jako kombinaci priorit byznysu a technických odhadů. Jakmile se skutečnost oproti plánu změní, plán zaktualizujeme.
- Malé verze – rychle uvedeme jednoduchý systém do provozu a pak uvolňujeme malé verze ve velmi krátkých cyklech.

- Metafora (nám již známé stories nebo scénáře) – celý vývoj vedeme pomocí jednoduše sdíleného příběhu o tom, jak má celý systém fungovat z pohledu zákazníka.
- Jednoduchý návrh – systém by měl být navržen co nejjednodušeji, jak je v daný okamžik možné. Nepotřebnou komplikovanost odstraníme ihned, jakmile je zjištěna.
- Testování – programátoři neustále píší unit testy, které musí proběhnout bez chyby, proto, aby mohl vývoj pokračovat. Zákazníci píší funkční testy.
- Refaktoring – programátoři restrukturují systém beze změny jeho chování, aby odstranili redundance kódu, zjednodušili systém a zdokonalili komunikaci.
- Párové programování – všechen zdrojový kód píší společně dva programátoři na jednom počítači, aby byla zaručena jeho kvalita a nejvhodnější návrh.
- Společné vlastnictví – všichni mohou měnit jakýkoliv zdrojový text systému kdekoliv a kdykoliv, v týmu neexistují žádné ostrůvky znalostí, které má jen jeden z členů týmu.
- Nepřetržitá integrace – systém integrujeme a sestavujeme několikrát denně, když je nějaký úkol (scénář) dokončen, aby byla zaručena sestavitelnost řešení, bezchybnost a také abychom předešli integračním problémům.
- 40ti hodinový týden – pravidlem je, že nepracujeme déle než 40 hodin týdně. Pokud pracujeme přesčas, tak nikdy ne dva týdny za sebou. Člověk, který takto pracuje, je pak unavený a více chybuje.
- Zákazník na pracovišti – do týmu patří i skutečný a živý budoucí uživatel systému, který je k dispozici na plný úvazek, aby odpovídal na otázky.
- Standardy pro psaní zdrojového kódu – programátoři píší všechen zdrojový kód v souladu s pravidly, která zdůrazňují komunikaci prostřednictvím zdrojového kódu.

Jak již bylo řečeno, slabé stránky jednoho postupu je snažíme zakrýt silnými stránkami dalších postupů. Jak tedy může fungovat například pouze *hrubé plánování* na počátku projektu? Potřebujeme přeci vědět, co, jak a kdy budeme dělat. Nemůžeme přece pořád aktualizovat plán – trvalo by to dlouho a zákazníky by to znervózňovalo. Pokud ovšem nepoužijeme tato pravidla:

- Zákazníci by si aktualizovali plán sami na základě odhadů poskytovaných programátory.
- Zpočátku bychom měli mít dostatečný plán k tomu, abychom zákazníkům poskytli hrubou představu o tom, co je možné uskutečnit během přístích pár let (milníky).
- Verze bychom uvolňovali v krátkých intervalech, takže jakékoliv chyby plánu by měly dopad nejvýše několik málo týdnů či měsíců.
- Zákazník by byl součástí týmu, takže by mohl rychle spatřit potenciální změny a příležitosti ke zlepšení.

Pak lze začít uskutečňovat vývoj pomocí jednoduchého plánu, který bychom s postupem projektu neustále zdokonalovali, jak se zpřesňují naše odhady,

znalost projektu a odstraňují rizika. Jedná se tedy o aplikaci iterativně inkrementálních principů tak, jak je známe.

Stejnou otázku si budete určitě klást i v případě *jednoduchého návrhu*. Jak to může fungovat, vždyť pro kvalitní, stabilní a rozšiřitelný systém potřebujeme mít adekvátní návrh a vůbec nezačínat nejjednodušším řešením. Měli bychom navrhovat a navrhovat, čímž se však zaženeme do kouta, zastavíme se, neschopni systém jakkoliv dále rozvíjet. Opět platí ano, to by se mohlo stát, pokud však nepoužijeme následující pravidla:

- Jsme zvyklí na refaktoring, který je ošetřen unit testy, takže provádění změn nedává důvod k obavám.
- Máme jasnou celkovou metaforu, takže máme jistotu, že další změny návrhu budou mít tendenci ji následovat.
- Programujeme v páru, čímž máme důvěru, že děláme jednoduchý a nikoli stupidní návrh.

Tak bychom mohli demonstrovat u všech 12ti principů, jak slabinu jednoho principu zakrývají silné stránky ostatních principů. Proto je zřejmé, že každý princip zvlášť může v určité oblasti pomoci, ale jejich síla a tedy i síla XP je v neustálém dodržování všech 12ti principů, kterými samozřejmě prostupují čtyři základní hodnoty.

Na závěr můžeme dodat, že právě XP výborně doplňuje Scrum, jelikož poskytuje vývojářům inženýrské praktiky a postupy, které ve Scrum chybějí. Tato kombinace také v praxi opravdu velmi dobře funguje. V praxi a IT komunitě jsou diskutabilní některé praktiky XP, zvláště onsite zákazník či párová práce 40 hodin týdně. Níže v textu rozebereme vhodné způsoby aplikace párové práce tak, abychom využili výhod této praktiky, ale aby zároveň příliš nabourávala osobní prostor a tempo vývojářů. Praktika onsite zákazníka je v dnešní ekonomicky turbulentní době problém, zvláště u menších společností, které si nemohou dovolit platit svého zaměstnance-experta za to, že nepracuje, ale pomáhá vývojovému týmu. Je však nutné říci, že takový člověk nediskutuje celých 40 hodin týdně s týmem, takže může plnit také svoje úkoly. Možnou variantou je rotace této role mezi více lidmi od zákazníka či její omezení například na 10-20 hodin týdně. Myšlenkou za touto praktikou je však možnost okamžitě vyjasnit vznikající nepochopení, často demonstrovat výstup apod. Omezení doby, kdy je zákazník s týmem, by tedy nemělo v žádném případě limitovat spolupráci a zpětnou vazbu mezi týmem a zákazníkem. Náklad na jednoho zaměstnance zákazníka, který dočasně žije s vývojovým týmem se většinou bohatě vrátí díky výslednému systému, který dělá co má, je kvalitní a přispívá k větší efektivitě, než kdyby úzká spolupráce s týmem vývoje neexistovala.

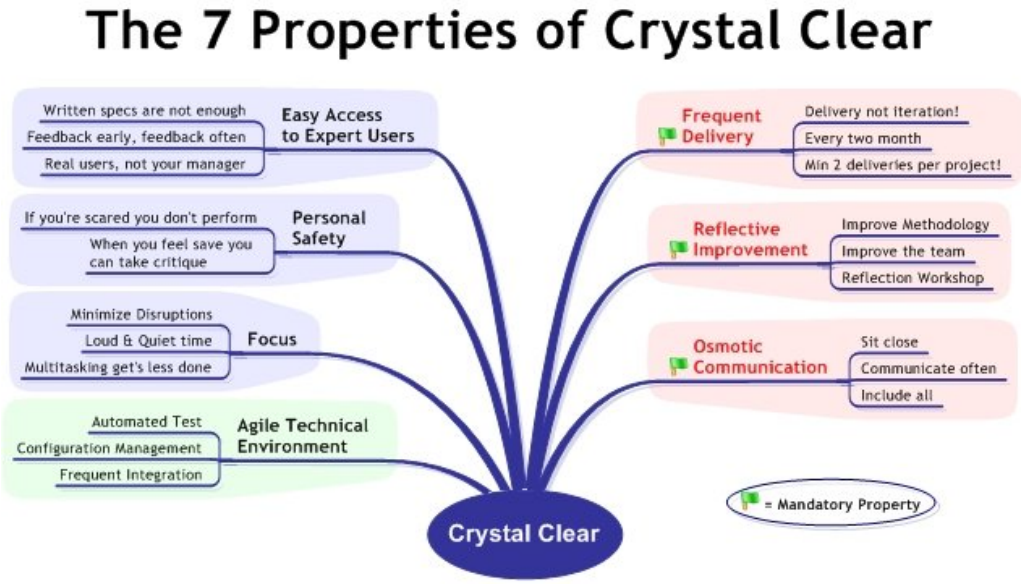
2.3 Další agilní přístupy

Scrum a XP nejsou jediné agilní přístupy, ale v praxi jsou nejvíce rozšířeny, proto jsme si je popsaly více do detailu. Z dalších agilních přístupů ještě zmíníme ***rodinu metodik Crystal***, jejichž autorem je další respektovaná osoba v oblasti agilního vývoje software, Alistair Cockburn. Crystal opět není metodika, ale opět spíše návod, jak si podle kontextu projektu a jeho omezení

vlastní metodiku vytvořit. Hlavní zaměření je stejně jako u ostatních agilních přístupů především na lidi, spíše než na procesy či technologie.

Sám Alistair Cockburn definuje Crystal přístup následovně [Co04]: „Je to vlastně velmi lehké, jediné, co potřebujete je následující:

- „team leadera – architekta a dva až sedm dalších vývojářů,
- sedících v jedné velké či v přilehlých místnostech,
- používajících tabuli a flipchart,
- mající snadný přístup k uživatelům (expertům),
- kteří nejsou vyrušování a doručují běžící, testovaný kód uživatelům,
- každý měsíc nebo dva, maximálně čtvrtletně,
- reflektující a přizpůsobující se neustále, pravidelně.“



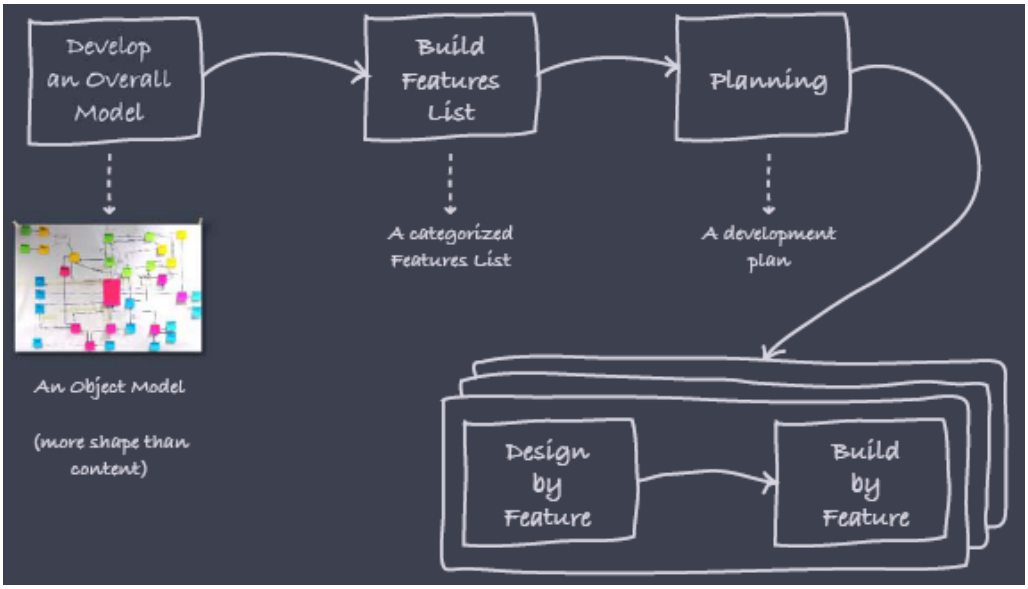
Obr. 2-6: Myšlenková mapa ukazující zdroje a vlastnosti jednotlivých vlastností Crystal přístupu. (zdroj: web wissel.net [Wi05]).

Pojďmě si tuto myšlenkovou mapu probrat krok po kroku. Crystal definuje bezpečné vlastnosti, kdy první tři jsou vyžadovány a další čtyři pomáhají týmu posunout se více do „bezpečné zóny“. Jedná se o následující [Co04]:

1. časté dodávky (*frequent delivery*),
2. zlepšování s rozmyslem (*reflective improvement*),
3. osmotická komunikace (*osmotic communication*),
4. osobní bezpečí, jistota (*personal safety*),
5. soustředení, zájem (*focus*),
6. snadný přístup k uživatelům, expertům (*easy access to expert users*),
7. prostředí s automatickými testy, konfiguračním managementem a neustálou integrací (*A Technical Environment with Automated Tests, Configuration Management, and Frequent Integration*).

Posledním z agilních přístupů, který by měl zaznít je ***FDD (Feature Driven Development)***, česky vývoj řízený rysy, vlastnostmi. Autory tohoto přístupu jsou Jeff De Luca a Peter Cod. Myslím, že pokud má čtenář alespoň trochu obrázek o problematice objektového programování a objektově orientované

analýzy a návrhu, pak není třeba více zmiňovat. My však přece jen něco o FDD řekneme. Tento přístup vlastně staví na důležitém stavebním kameni agilních přístupů, jímž je práce s požadavky (tzv. *scope management*) a pochopení potřeb. Základní myšlenkou je, že hlavní rysy (*features*) jsou klíčové, základní prvky vývoje software. Sled základních aktivit, které je třeba vykonávat podle FDD a jejich vazby popisuje následující obrázek:



Obr. 2-7: Aktivity FDD a jejich sled (zdroj: [Lu05])

FDD je modely řízený (*model driven*) iterativní přístup skládající se z pěti aktivit. Pro řízení postupu prací a kontrolu projektu jsou definovány milníky pro každý rys, vlastnost. Pět základních aktivit podle FDD je [Lu05]:

1. Vytvoření globálního modelu (aktivita *Develop Overall Model*) – identifikace obsahu projektu a kontextu, tvorba (sestavení z existujících) globálního doménového modelu.
2. Sestavení seznamu rysů, vlastností (aktivita *Build Feature List*) – podle znalosti nabyté v předchozím kroku sestavíme priorizovaný seznam rysů ve formě <akce> <výsledek> <objekt> př.: Spočítej celkový součet prodejů. Tato aktivita by neměla trvat více než 2 týdny.
3. Plánování (aktivita *Planning*) – vytvoření vývojového plánu a přiřazení práce vývojářům.
4. Návrh podle rysu (aktivita *Design by Feature*) – detailní design pouze pro vybrané rysy (implementovatelné ve 2 týdnech), na konci návrhu proběhne inspekce.
5. Sestavení podle rysu (aktivita *Build by Feature*) – daný rys je naimplementován, proběhne unit testování, inspekce kódu a poté je kód sestaven.

Tolik základní principy a odlišnosti agilních přístupů. Pokud vás tato problematika zajímá více, lze doporučit česky psanouliteraturu od V. Kadlece nazvanou Agilní programování, vydána v Computer Press v roce 2004.

V další kapitole se podíváme na mylné názory a představy o tom, co to agile je a není, jelikož tato diskuse může čtenářům upravit pochopení, které o agilních

přístupech mají. Dále detailněji prodiskutujeme vybrané agilní praktiky a techniky, které jsou důležité nejen z hlediska praxe a zavedení agilních principů, ale také pro zvládnutí předmětu Ročníkový projekt.

2.4 Mylné představy a názory o agile[1]

Stejně tak jako války a konflikty plynou často z neznalosti, strachu z nového či z hájení starých postupů (např. vodopádu) a pravd (ano, země je opravdu kulatá ☺), tak i nové přístupy k vývoji software nemají na růžích ustláno, i když jsou podpořeny fakty a argumenty ve formě úspěšných projektů. K základním lidským stránkám bohužel patří strach z neznámého, dříve to byly hromy a blesky, nyní jsou to nové přístupy. Další naší stránkou je rezistence vůči změnám, zvláště pokud jsou nám vnuceny a nemůžeme se rozhodnout sami. Určité typy lidí jen nerady opouštějí zažité postupy, které umí a mohou dále zdokonalovat nebo se naopak již nemusí učit nové věci. Taková strategie je v IT docela problém, jelikož vývoj jde velmi rychle dopředu a to i v oblasti vývoje SW. Jsou ale také lidé, kterým nevyřešené problémy nenechají spát, rádi se neustále vzdělávají, staví se výzvám. Právě pár takových stálo u zrodu Agilního manifestu, viz výše. Následující kapitola stručně shrnuje nepravdy o agilních přístupech a nepochopení, plynoucí většinou z tradičního způsobu myšlení. Diskutované mýty a nepravdy jsou následující:

- V agilních přístupech se nedokumentuje.
- Nenavrhujeme žádnou architekturu.
- Jen vývojáři mají/musí být agile.
- Podpora managementu firmy není třeba.
- Metodika

V agile se nedokumentuje

Prvním běžným tvrzením je, že v agilních přístupech se nedokumentuje, neexistuje žádná dokumentace, modely, ale pouze kód. Toto tvrzení není pravdivé a plyne ze špatné interpetace principu: *working software over comprehensive documentation*. V Agile opravdu modelujeme, dokumentujeme, ale pouze v jiné formě a pouze to, co je nutné. Use case či user stories popisují funkčnost, ale (zde jasné či nepodstatné) detaily jsou zachyceny v unit testech či test casech funkčních. Další dokumentací je samozřejmě dobře strukturovaný a komentovaný zdrojový kód, který je dle výzkumů jediná používaná dokumentace při provozu a údržbě SW [dS05], [dS07]. Modely také existují, ale například pouze jako náčrty na tabuli či na papíře. Jelikož jsou běžné krátké releasy, kód se často mění (refaktoruje), stálo by přepracování modelů či jiných dokumentů spoustu úsilí nebo by se tyto staly brzy neaktuální. Proto se snažíme dokumentaci automatizovat a mít ji pouze na jednom místě (unit testy, zdrojový kód) a dokumentujeme na vyšší úrovni: rozhraní a architekturu, které se týká další mýtus.

Nenavrhujeme žádnou architekturu

Druhý typický mýtus je, že v Agile nenavrhujeme žádnou architekturu. Opět se jedná o nepochopení principu a zde také způsobu práce. Je pravdou, že na

[1] Článek byl autorem (J. Procházka) publikován v Systémové integraci č. 3, ročník 2008.

začátku projektu si nesedneme ke všem požadavkům, jelikož je ani nemáme a neřešíme „papírovou architekturu“. V úvodních iteracích však řešíme určitou kostru aplikace, vrstvy, způsob ukládání dat, komunikaci s ohledem na celé řešení. Definujeme rozhraní vrstev, jelikož toto mohou být zásadní technické problémy (např. integrace s bankovním legacy systémem, pravidelné dávkové importy, implementace standardů), které mohou silně ovlivnit celé řešení – použitou technologii, způsob komunikace mezi vrstvami, nutnost prototypů apod. ***Toto vše ale ověřujeme implementací nejdůležitějších scénářů***, neřešíme jen návrhové problémy, ale návrhové problémy v kontextu nejdůležitějších scénářů. Takže ***implementace základních scénářů nám určuje architekturu***. V tomto malém rozdílu je skryto nepochopení. Detailní architekturou se zabýváme vždy jen pro danou iteraci, tj. jen pro scénáře, které nyní implementujeme. Neřešíme architekturu s ohledem na požadavky, které bychom měli implementovat v budoucích iteracích, protože se mohou změnit priority, tyto požadavky se mohou posunout do dalších releasů či úplně zrušit a naše snaha může přijít vniveč. Neděláme tedy kompletní architekturu do šuplíku, ale vždy jen pro aktuální potřebu v kontextu nejdůležitějších scénářů.

Jen vývojáři jsou/mají být agile

Jen developeři jsou/mají být agile, obchodníci a management nemusí být – toto je další z problémů a nepochopení, i když ne tolik zmiňovaný jako ostatní. Jeho dopad je však stejně zásadní. Pokud zákazníkovi řekneme (tedy spíše naši obchodníci), že jsme schopni dodat opravdu všechno, co si vymyslel, za tu cenu, v tom čase a v té kvalitě (což se opravdu často děje), bez ohledu na to, co říká trojúhelník kvality, pak nás samozřejmě ani agile nezachrání. Pokud zákazník nebude chtít spolupracovat, nebude se chtít účastnit pravidelných demonstrací aplikace, kde si hraje s dosud vyvinutým řešením, komentuje je a připomínkuje, nelze očekávat úspěch. Musíme zákazníka učit, vysvětlovat mu, proč je nutné vidět aplikaci a korigovat své i naše představy, že jen tak dostane opravdu to, co očekával (viz Wegnerova lema[2]). Je to těžké, protože zákazník byl po léta zvyklý na začátku nadiktovat všechny, tedy i ty nepotřebné požadavky (kterých je podle Standish Group Chaos reportu [SG] více než polovina) a na konci očekával řešení (to už ale víme, že nefunguje), jehož přínos a také kvalita jsou mnohdy diskutabilní. Toto téma bylo diskutováno v textu Informační systémy 1.

Podpora managementu firmy není třeba

Podpora managementu firmy není třeba, stačí, aby mi jako vývojáři řekli: ano, nějak si to teda udělej (rozuměj zaveď agile) – toto je další častou chybou a naivní představou. Co je ještě horší než slabá podpora managementu, je zavádět agile praktiky a životní cyklus potají. Tajné zavádění či nedostatečná podpora ze strany managementu v podstatě úplně zhatí celou snahu, jelikož bez podpory managementu (který má tuto vizi protlačit) a bez oficiálních prostředků a schválení (vývojáři absolvují tréninky, zpomalí se tempo vývoje) agile přístup ani zavést nelze, jedná se totiž o úplnou změnu chování a myšlení všech lidí v organizaci a také o změnu vystupování směrem k zákazníkovi, což

[2] Wegnerova Lemma nám říká, že nejsme schopni kompletně popsat interaktivní systém, náš mozek to prostě neumí [We97]. Originální citace lemy zní: “*It is impossible to fully specify or test an interactive system designed to respond to external inputs.*”

potají ani moc dobře nejde. Standish Group ne nadarmo řadí podporu managementu (*executive support*) na 1. místo svého pravidleného Chaos reportu [SG], který popisuje situaci ohledně projektů v IT. Pro zdůraznění důležitosti této podpory ještě zmíníme že na 2. místo odsunul dosud vedoucí angažovanost uživatelů v projektu.

Metodika

Představa, že agilní přístupy Scrum, XP, FDD či OpenUP jsou metodiky je dalším omylem. Jedná se totiž většinou o procesní frameworky (Scrum, OpenUP, RUP) či sady best practices. Metodika přesně říká co, kdy, kdo a jak má dělat. Když nevím co teď, mrknu do knihy na stranu 216 a přečtu si to. Procesní frameworky jsou jiné, popisují jaké kroky je možné provádět při vývoji software, jaké artefakty mohou zachycovat různé informace, prostě jaké činnosti se v praxi osvědčily. Na nás pak je, abychom si podle technologie, domény a znalostí týmu vybrali to, co konkrétně nyní potřebujeme, uznáme za vhodné dělat, tj. vytvořili si vlastní proces. Je to pochopitelné, každá organizace je jiná, má jinou strukturu, kulturu, distribuce týmu se různí, členové týmu mají jiné zkušenosti a znalosti, jejich povahové rysy jsou různé, také zákazník se chová jinak. Proto není možné postupovat vždy podle stejné či podobné metodiky, ale je třeba si daný proces vývoje upravit podle potřeb daného projektu (přesně toto doporučují či říkají dané přístupy). Toto základní nepochopení pak způsobí, že tým vytváří všechny artefakty a provádí všechny činnosti a vývoj se stane pomalých, chybovým a neefektivním. To vše jen proto, že si tým nevytvořil svůj proces jako podmnožinu popsaných aktivit, rolí a artefaktů. Z ***procesního frameworku si tedy pouze vybereme potřebnou podmnožinu, ale je třeba dodržet a následovat určité zásady a principy***. Funguje to stejně jako s bufetem ve formě švédských stolů. Bereme si jen to, co máme rádi, s čím máme dobré zkušenosti, sem tam zkusíme něco nového a snažíme se tomu přijít na chuť, ale musíme dodržovat určité zásady. Jídlo dáváme na talíř, většinou jíme příbory, nápoje naléváme do skleniček apod. Tento mýtus zní jako nedůležitý a spíše teoretický problém, prakticky však může stát za neúspěšnou implementací.

2.5 Možné problémy – na co si dát pozor

Nyní víme, co je běžně dezinterpretováno či nepochopeno a dáváno jako slabina agile přístupů. Jaké jsou však skutečné možné problémy? Na co si dát pozor? Stručně by se dalo říci, že agile:

- je hlavně o lidech (jako všechno při vývoji SW),
- vyžaduje neustálé učení a zlepšování (jako všechno při vývoji SW),
- vyžaduje podporu shora také aktivní přístup zákazníka.

Nejzásadnější u celého vývoje SW je snaha a ochota lidí spolupracovat, nebát se změn a nových věcí, nezříkat se odpovědnosti (samozřejmě pokud máme i rozhodovací pravomoci), být inovativní (jak ty principy vlastně implementovat v našem projektu?). Být otevřený novým věcem je kritické. S tradičním způsobem myšlení nejsme schopni agile či iterativně inkrementální principy pochopit správně, natož implementovat a neustále zlepšovat. Spolupráce lidí je bezpodmínečná v rámci celého vývoje pro správné pochopení a zachycení požadavků od zákazníka, jejich neustálá komunikace a vysvětlování

vývojářům, testování těchto požadavků testery, demonstrace zákazníkovi a pochopení jeho případných zlepšení, námitek apod.

Dalším problémem agilních přístupů bývá specifikace požadavků. Specifikace požadavků v XP či Scrumu je založena na user stories (viz skripta Softwarové inženýrství, kapitola Specifikace požadavků) místo na use case. User stories jsou jakési odlehčené verze scénářů, mnohdy jejich fragmenty. Problémem této odlehčené verze je tedy chybějící celistvý pohled na systém. Stories jsou atomické jednotky chování, proto zobrazení všech lze jen v jakémsi zásobníku práce (např. tabulka, wiki), neexistuje přehledné grafické vysokoúrovňové zobrazení. Chybějící celkový pohled pak může způsobit opomenutí důležitých funkčností, vazeb a návazností a vést ke špatnému pochopení a implementaci požadavků. Je třeba říci, že Scrum a XP s tímto problémem pracují a přinesly tzv. epiky a témata, která sdružují stories do skupin podle jejich příbuznosti.

Díky zažitému způsobu myšlení si lidé pracující na projektech často myslí, že pracují agilně, říkají: *„Ano my máme iterace:*

- *1. iterace požadavky,*
- *2. iterace analýza,*
- *3. iterace implementace atd“*.

Nebo říkají: *„Ano, jsme iterativní, jedeme podle RUP.“* A po 4 měsících pořád dělají specifikace požadavků. My však ale víme, že obě tato sdělení jasně ukazují vodopádový způsob práce. Agilní či iterativně inkrementální projekty mají krátké iterace (2 týdny až 1 měsíc) a v každé iteraci produkují část spustitelné aplikace, které se pak na sebe nabalují.

Jedním z dalších možných problémů, které mohou způsobit težkosti při vývoji jsou chybějící striktní evaluační kritéria, či přesně vymezené požadavky pro danou iteraci. Velmi běžně se můžeme setkat s validačními kritérii typu:

- kvalitní UI,
- hotová implementace,
- vyhotovená uživatelská dokumentace,
- a podobné, velmi vágní ustanovení.

Tato vágní sdělení jsou často zdrojem problémů. Co si máme představit po pojmem kvalitní UI??? Každý uživatel bude za kvalitní považovat něco jiného, navíc, znamená to implementaci nějakého standardu nebo akceptace nějakými testy? Nic z toho není zřejmé a proto může být vnímáno různými stranami zcela odlišně. Pokud není zákazník spokojen, může dodavatele popotahovat a nechat si udělat spoustu práce zdarma. Vágní evaluační kritéria mohou vést až k soudním přím. Lepší je tedy mít striktně definovaná objektivní evaluační kritéria, která nelze interpretovat jinak, např:

- 100% unit testů proběhlo úspěšně,
- 100% integračních a systémových testů proběhlo úspěšně,
- 96% systémových testů prošlo úspěšně,
- Současná verze neobsahuje více než 1 kritickou chybu,
- Současná verze neobsahuje více než 2 středně kritické chyby,
- data jsou uložena v DB ve stejném formátu jako byla zadána,

- zákazník akceptoval implementované scénáře v dané iteraci,
- vyhotoveny tréninkové materiály pokrývající aktuálně implementované funkčnosti/scénáře.

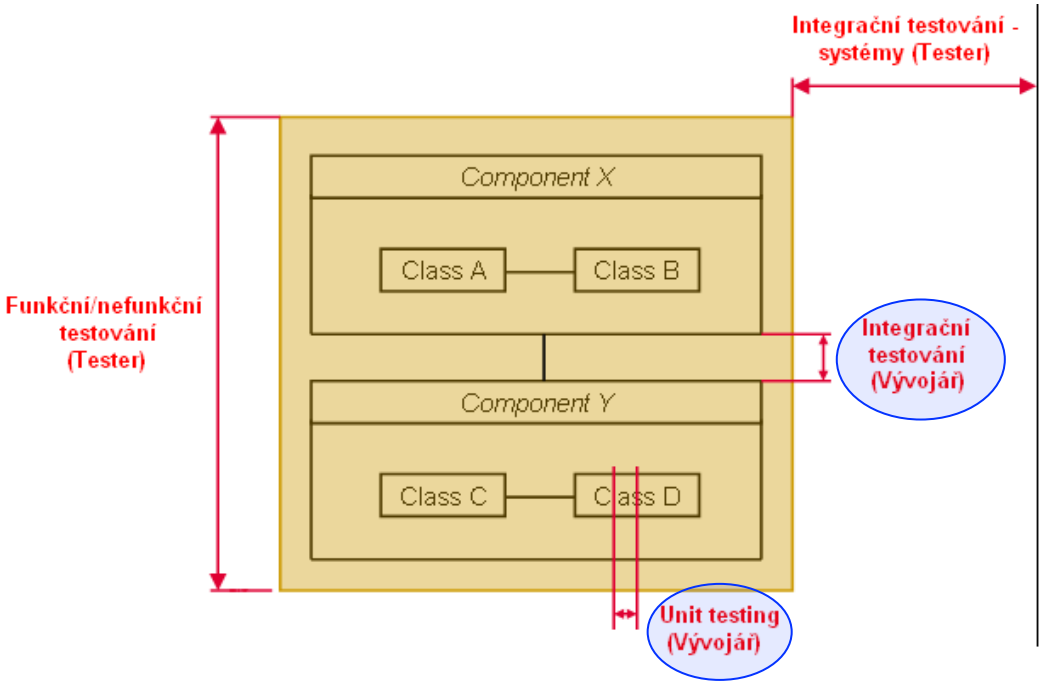
Pokud si uvědomíme, že tato kritéria používáme k ohodnocení úspěchu a neúspěchu iterace (zda je splněna či ne) a podle toho například dostáváme zaplaceno, je nasnadě, že zákazník může tvrdit opak toho, co my, tj., že kritéria nejsou naplněna, což vágní interpretace na rozdíl od objektivní umožňuje. Pokud máme jako argument report ukazující 100% úspěšnost unit testů, o polemice nemůže být ani řeč. Pokud jsme placeni od iterace, mohou objektivní evaluační kritéria na konci každé iterace nutit zákazníka více spolupracovat, více se aplikaci věnovat a podat zpětnou vazbu.

2.6 Agilní praktiky a techniky

Většina výše zmíněných agilních přístupů používá jisté praktiky a techniky, které pomáhají zavádět agilní principy do praxe a také umožňují efektivnější a více kvalitní vývoj software. Kromě praktik podporujících komunikaci a změnu řízení, včetně fixování proměnných trojúhelníku kvality, existují také další praktiky a techniky. Jedná se především o znovu použití komponent, subsystémů (např. Hibernate, Elipse, COTS – Commercial-Off-The-Shelf) software, testy řízený vývoj, denní krátké meetingy, retrospektivy na konci iterace/sprintu, neustálá integrace a buildování výsledného produktu či párové programování. Následující kapitoly popisují tyto praktiky detailněji a ukazují jejich použití v praktickém kontextu.

2.6.1 Test Driven Development (TDD)

Každý kód, který v rámci vývoje či údržby vytvoříme či změníme, je nutné otestovat. Aby člověk zlepšil svoje chování (např. způsob programování) a neopakoval stejné chyby, prostě se dokázal zlepšovat, potřebuje ***jistou formu zpětné vazby***. Rychlost zpětné vazby je při učení kritická. Pokud budeme vždy čekat na to, až výsledek naší práce otestují testeři či dokonce uživatelé, nespojíme si chybu s naším konkrétním chováním a nemůžeme se zlepšit. Z důvodu rychlé zpětné vazby a také z důvodu zvýšení kvality napsaného kódu používáme něco, čemu se říká vývojářské testování (tzv. *unit testy*). Ano, vývojáři musí testovat svoji práci také. Vývojářské testování může být dokonce stejně tak zábavné, jako samotné psaní zdrojového kódu. Co má vývojář testovat a na jaké úrovni shrnuje následující obrázek.



Obr. 2-8: Rozsah testování a odpovědnosti vývojářů

Vývojářské (*unit*) testování je psaní testů ve formě zdrojového kódu na úrovni nejmenších jednotek kódu (tříd, modulů, webových služeb). Pokud mluvíme o vývoji řízeném testy (*Test Driven Development* - TDD), pak musí platit zásada, že nejdříve napíšeme testy a až potom samotnou implementaci funkčnosti. Tuto techniku uplatníme v oblasti údržby při implementaci nových funkčností a změně stávajících. Princip tohoto přístupu, který pochází z Extrémního programování (XP) [Be99], je velmi jednoduchý. Nejdříve napíšeme vývojářský (*unit*) test a až poté vlastní kód hledaného řešení. Každého asi nyní napadne otázka: *„Jak můžeme psát test, když ještě nemáme vlastní implementaci dané funkčnosti? Jak vím, co mám testovat, když ještě nevím, jak se to chová, co to dělá?“* Přesně v těchto otázkách je ukryt smysl testy řízeného programování. Význam psaní testů před kódem je v tom, že ***v průběhu psaní testů si rozmýšlíme vlastní kód, vlastní návrh*** a většinou nás napadne ***několik (kvalitnějších) řešení***, která by nás nenapadla, kdybychom začali programovat hned po analýza a (doufejme že i) návrhu. Výhod má tento přístup několik [Be99]:

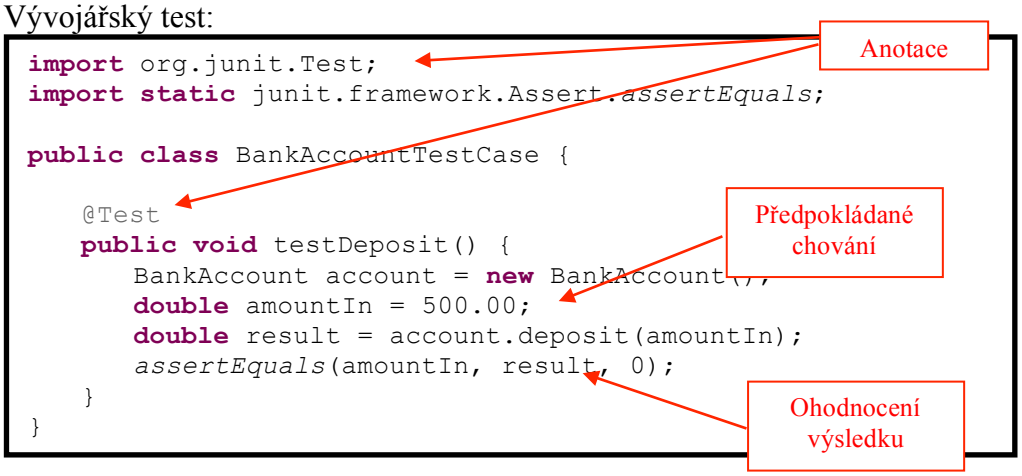
- výběr nejvhodnějšího implementačního řešení – při psaní testů promyslíme několik variant a můžeme vyberat podle nás tu nejvhodnější (tedy ne první, co nás napadne),
- ihned po napsání zdrojového kódu jsme schopni ověřit funkčnost kódu vývojářským (*unit*) testem,
- máme okamžitou zpětnou vazbu o funkčnosti a kvalitě kódu, díky čemuž se můžeme učit a okamžitě zlepšovat, neopakovat stejné chyby v dalším napsaném kódu,
- v případě hledání chyby (případ údržby a opravy chyby) máme nástroj pro lokalizaci chyby, snadné nalezení místa s chybou,
- v případě refaktoringu, implementace změny či rozšíření okamžitě vidíme, jestli jsme nezanesli do programu chybu (což je využití hlavně v našem kontextu podpory a údržby).

Postup podle TDD je následující:

1) Vytvoříme nový test – test nemůže projít, jelikož neexistuje implementace, kterou má testovat; důležité je mít pochopení problematiky (požadavků) např. z detailních scénářů.
2) Spuštění všech unit testů a zjištění, že nový nebyl úspěšný – ověření, že nový nemůže projít, jelikož ještě neexistuje kód, který má testovat; ověření jeho správnosti (odhalení případné chyby v testu).
3) Vytvoření nějakého kódu – napsání hrubého kódu za účelem úspěšného průchodu testem, nejedná se o finální implementaci!
4) Spuštění automatických testů a zjištění úspěšného průchodu novým testem – kód splňuje požadavky, můžeme začít psát finální implementaci.
5) Refaktoring kódu – zlepšení kvality kódu, jeho vyčištění, doplnění funkcionality, nahrazení kouzelných čísel či řetězců v kódu; zanesení chyby je kontrolováno spouštěním automatických testů.
6) Opakování postupu – iterativně tento postup opakujeme, jak přidáváme nové funkčnosti, opravujeme chyby či implementujeme změny.

Následující ukázka vývojářského (*unit*) testu a kódu ukazuje detail a vytvoření testu a následně vlastní implementaci v jazyce Java. Příklad ukazuje také spuštění a výsledky testování pomocí tohoto vývojářského testu v integrovaném nástroji JUnit.

Vývojářský test:

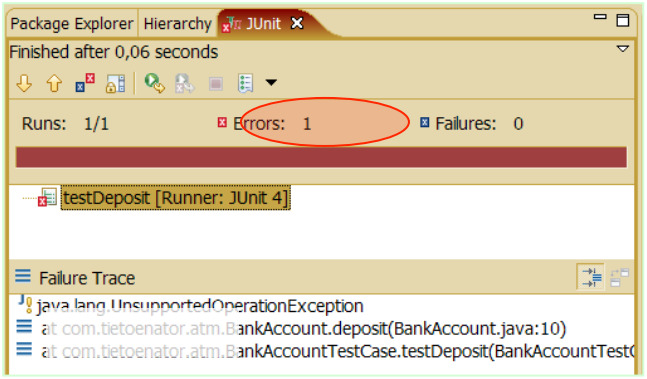



```
import org.junit.Test;
import static junit.framework.Assert.assertEquals;

public class BankAccountTestCase {

    @Test
    public void testDeposit() {
        BankAccount account = new BankAccount();
        double amountIn = 500.00;
        double result = account.deposit(amountIn);
        assertEquals(amountIn, result, 0);
    }
}
```

Implementace funkčnosti pro daný test:

```
public class BankAccount {

  public double deposit(double amount) {
   throw new UnsupportedOperationException("Not implemented
                          yet.");
  }
}
```

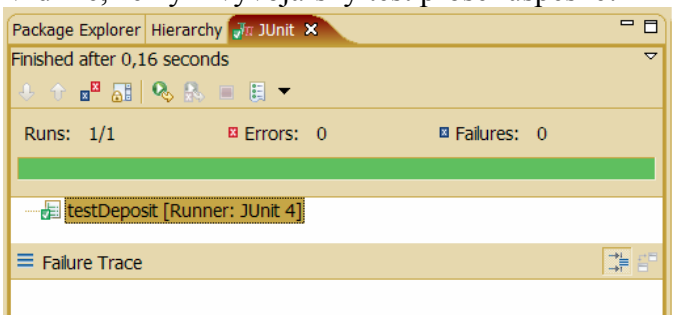
Spuštění výsledku, výborně, nefunguje, jelikož ještě neexistuje smysluplná implementace, resp. vrací danou výjimku:



Provedeme nyní nejjednodušší možnou implementaci (tzv. *quick and dirty*):

```
public class BankAccount {

    public double deposit(double amount) {
        return amount;
    }
}
```

Vidíme, že nyní vývojářský test prošel úspěšně!



Stejně pokračujeme pro další metody implementované funkčnosti. Nakonec ještě nezapomeneme udělat refaktoring prvotní (*quick and dirty*) implementace do čitelného a rozšiřitelného řešení následující konvence pro psaní kódu a zásady objektového programování. Typicky je vhodné nekontrolovat pomocí unit testu pouze správné chování pro správné vstupy, ale ošetřit (ve dalších testech) i nestandardní vstupy nebo hodnoty (záporný věk, hodnoty *null apod.*) a zajistit ta korektní chování i pro ně.

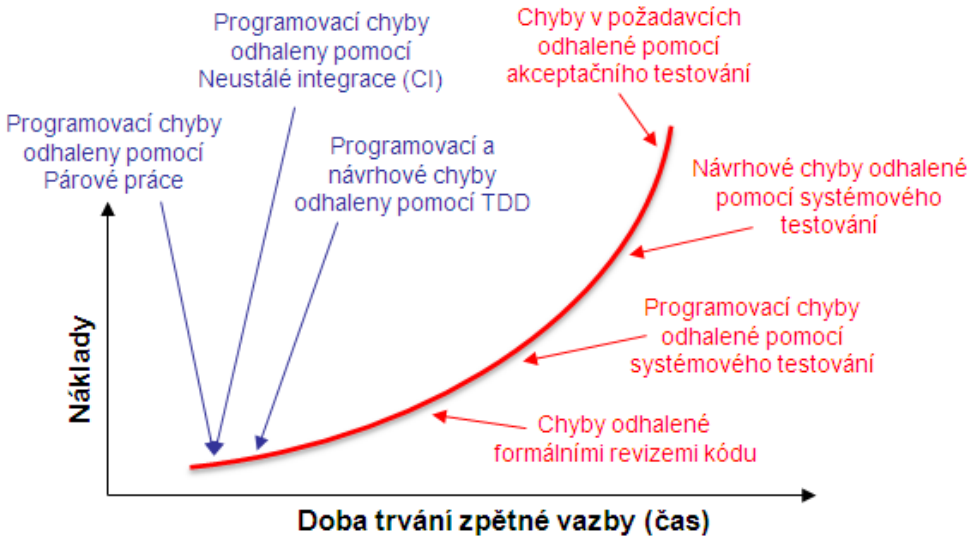
Pro vývojářské testování je možné využít různé automatizované nástroje či frameworky. Nejznámějším je pravděpodobně xUnit, který existuje ve variantách pro různé programovací jazyky, např. původní verze SUnit pro Smalltalk, dále JUnit pro Javu, NUnit pro .NET, PHPUnit pro PHP, CUnit pro jazyk C apod.

Můžete si pomyslet, že je jedná pouze o hezkou teorii z knihy. Navíc v praxi musí její aplikace spotřebovat nějaký čas, když kromě vlastní implementace musíte napsat a promyslet i vývojářský test a promyslet vhodná řešení. Normálně generujete řešení po prvotní analýze a návrhu a většinou je to to nejlepší, minimálně to nějak funguje doteď a nemáte s údržbou tohoto kódu zásadní problémy (až na ty nezbytné občasné chyby). Ano, použitím ***TDD***

přístupu samozřejmě trvá programování stejného požadavku přibližně o 10-20% déle. V počátcích, kdy se tým učí psát vývojářské testy, může být toto číslo vyšší, později klesá. Ale vlastní testování v průběhu vývoje či změny, identifikace, lokalizace a oprava chyby je o mnoho rychlejší (díky využití nástrojů a aktuální znalosti kontextu toho, co implementuji). Dalším obrovským přínosem je možnost okamžité zpětné vazby, ne až po pár týdnech, kdy někdo reportuje incident či chybu, a z toho plynoucí neustálé učení. Programátor se ***okamžitě poučí o své chybě či nevhodnosti návrhu*** a hned v příští metodě může dané poznatky aplikovat. Výsledný zdrojový kód je tedy inkrementálně kvalitnější a daný programátor se zlepšuje a to i kontextu jednoho dne.

Když se na TDD podíváme ***v kontextu celého životního cyklu softwaru*** (tedy od vývoje až po provoz, který trvá o mnoho déle než vlastní vývoj), ***je*** těch ***10-20% času navíc nejlepší možnou investicí***. V celkovém kontextu životního cyklu softwaru se nám daná investice vrátí velmi brzy v provozu. Pokuste se tedy jako součást týmu vývoje či implementace změn oprostit od hranic vašeho oddělení a myslete opravdu systémově (i na provoz a údržbu)!

Situaci také shrnuje následující obrázek. Křivka ukazuje rostoucí cenu chyby, respektive cenu jejího odstranění v průběhu životního cyklu softwaru. Je zřejmé, že tato cena exponenciálně roste, čím více se blížíme ostrému provozu. Co to znamená? Jak to vysvětlíme? Pokud udělá programátor chybu či zvolí nevhodný návrh (špatná technika, neuvolnění zdrojů, zapomenutá kontrola na *null* objekt, ...) a ta je ihned odhalena kolegou při párovém programování či zachycena vývojářským (*unit*) testem, může být během několika minut odstraněna. Programátor zná kontext, má v paměti, co a proč dělal, má daný kód takzvaně „rozdělaný“. Kdyby se tato chyba projevila až ***v ostrém provozu, bude cena a čas opravy o dost vyšší*** z několika následujících důvodů. Uživatel nemůže pracovat s daným softwarem (např. zpracovávat objednávky), což může podniku generovat ztráty (resp. nepřinášet zisk), uživatel musí chybu reportovat a popsat; někdo na straně IT ji musí ohodnotit, přidělit k řešení, programátor ji musí odhalit, pochopit strukturu kódu a jeho význam, musí řešení otestovat a schválit zákazníkem a poté opět nasadit do provozního prostředí v čase k tomu určeném. Čas a úsilí těchto lidí stojí peníze a někdo je musí zaplatit, proto je odhalení a opravení chyby v provozním prostředí o hodně dražší než například TDD.



Obr. 2-9: Cena chyby v průběhu životního cyklu aplikace (červená křivka) a techniky minimalizující cenu opravy chyby v různých fázích životního cyklu aplikace. Všimněte si ceny TDD (zdroj [Am06]).

Pokud se zaměříme na prevenci zanesení chyb či jejich brzké odhalení již v průběhu implementace, i když se to může zdát z pohledu vývoje dražší[3]. Můžeme významně snížit cenu nejen tvorby, ale hlavně provozu a údržby softwaru. Současně s finančními aspekty tímto krokem zvýšíte její kvalitu a nepřímo spokojenost uživatelů a zákazníka.

Technika TDD je mnohdy programátory zavrhována, zvláště těmi zkušenými. Mnohdy proto, že ji nikdy nezkusili a tudíž mají strach z nového, nevěří metodě, nelíbí se jim více času nutného na tvorbu kódu včetně testů či jiné typologické důvody. Rozhodně jako každá nová věc vyžaduje i TDD učitý čas na její zvládnutí. Z osobní zkušenosti autora můžeme říct, že bychom při každé následující tvorbě programu nepostupovali jinak než podle TDD či alespoň s vývojářskými testy.

2.6.2 Refaktoring zdrojového kódu

Technika zvaná refaktoring kódu je jakýmkoliv zásahem do zdrojového kódu programu, jenž zlepšuje čitelnost kódu nebo jeho strukturu, to vše beze změny chování programu. Nedílnou součástí refaktoringu je vývojářské (*unit*) testování zajišťující zpětnou kontrolu, zda jsme změnou nezanesli do kódu chybu. Refaktoring neodstraňuje chyby, ani nepřidává novou funkcionalitu, cílem je zlepšit čitelnost kódu, jeho strukturu, zlepšit návrh, či odstranit části kódu, které se již nevyužívají. Inkrementálním zřetězením menších refaktoringů jsme schopni kompletně přepsat aplikaci, například změnit celý návrh z důvodu nevhodné architektury. Jelikož se chování aplikace nemění, uživatelé v průběhu refaktoringu nic nepoznají, výsledkem je pak stabilnější a

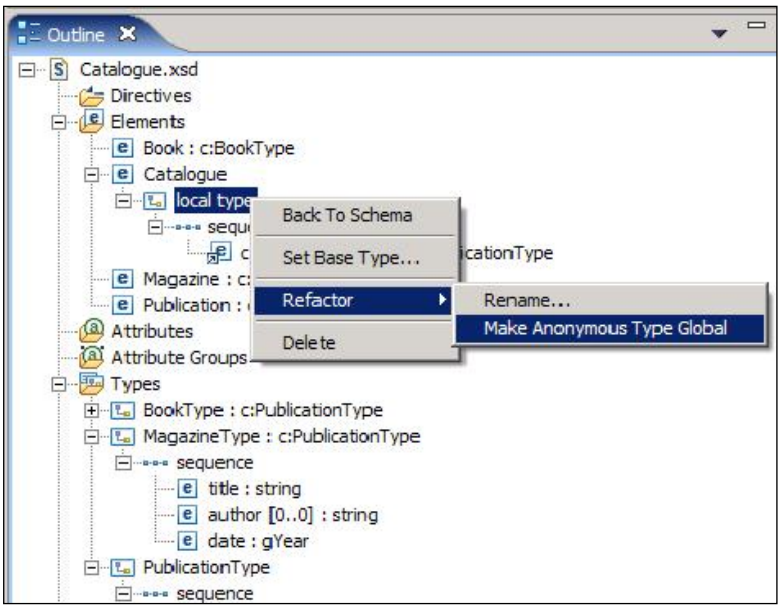
[3] Což je ukázka typické suboptimalizace a krátkodobého pohledu, velmi běžné v mnoha společnostech (jak malých, tak mezinárodních korporací): při vývoji ušetřím několik dní či týdnů (jednotky až desítky tisíc korun) na úkor kvality, testovatelnosti a rozšiřitelnosti řešení a v údržbě pak zákazník či poskytovatel platí stejnou a vyšší částku za zmírnění následků (oprava chyb návrhu či programování) ročně.

kvalitnější aplikace (to by naopak už měli uživatelé poznat). Kritickým faktorem tohoto přístupu jsou ale vytvořené a používané vývojářské (*unit*) testy, které spustíme **vždy všechny po každé změně**, abychom měli jistotu, že jsme nezanesli chybu do žádné z funkčností, které byly předmětem refaktoringu či jinam.

Místa vhodná k refaktorování většinou ***odhalíme pomocí symptomů*** jako jsou funkční třídy (příliš mnoho odpovědností nebo metod v jedné třídě), duplicitní kód, dlouhé metody a seznamy parametrů, datové shluky a podobně. Refaktoring můžeme rozdělit do několika skupin podle toho, čím se při něm zabýváme. Jedná se o následující:

- úpravy metod,
- přesouvání elementů mezi objekty,
- organizace dat,
- zjednodušení podmíněných příkazů,
- zjednodušení volání metod,
- generalizace.

Velkým pomocníkem při refaktoringu jsou IDE (*Integrated Development Environment*) nástroje. Umožňují automatizovat některé kroky prováděné v refaktoringu. Potřebujeme-li například změnit název metody či třídy, potřebujeme změnit také veškeré reference na tyto komponenty, což v podnikové aplikaci může znamenat desítky odkazů, pokud se jedná o rozhraní, tak i mnohem více. IDE nástroj nám umožní tento krok provést většinou velmi jednoduše, na jedno kliknutí.

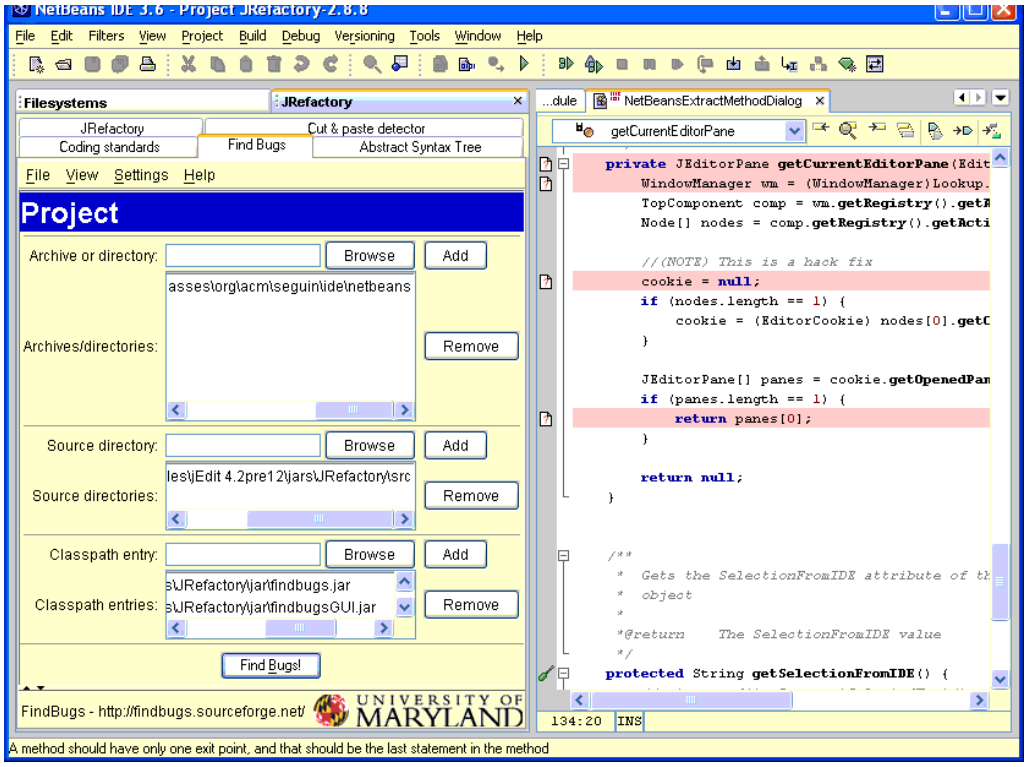


Obr. 2-10: Jednoduchost refaktoringu s podporou Elipse IDE (použití refaktorovací metody *Rename*…)

Nástroje sloužící k refaktoringu (většinou ve formě plug-inů do jednotlivých IDE) výrazně zjednodušují jeho provedení. Umožňují vykonávat jednotlivé akce, kroky mnohem rychleji, než v případě ručního refaktoringu, redukují množství chyb zanesených při refaktoringu a umožňují také identifikovat kód

vhodný k refaktoringu. Zvláště identifikace oblastí, které je vhodné refaktorovat se manuálně hledá obtížně. Příklad některých rysů, které umí refaktorovací nástroje:

- Přesun tříd mezi balíčky a přejmenování tříd.
- Přidání návrhových vzorů do kódu – přeskupení tříd (*abstract class*).
- Přesun metod, přejmenování parametrů.
- Extrakce nové metody ze zadaného bloku kódu.
- Kontrola konvencí zápisu kódu.
- Tvorba UML diagramů, které mohou sloužit k navigaci či zjednodušení složitosti, k pochopení kódu (velmi významné v údržbě).
- Nalezení chyb (pro běžné typy chyb jako uvolňování zdrojů, odkazování na *null* hodnoty apod.).
- Přidání *JavaDoc* komentářů do zdrojového kódu.
- Sběr metrik týkajících se zdrojového kódu.



Obr. 2-11: Nástroj JRefactory[4] a hlášení o chybě – metoda má mít pouze jedno místo návratu včetně zvýraznění oněch několika míst (zdroj: vlastní).

V kontextu vývoje a údržby je, stejně jako v případě TDD, vhodné zařadit refaktoring okolního kódu či třídy, pokud symptomy ukazují na porušená pravidla softwarové architektury či konvencí kódu v následujících případech:

- pokud implementujete požadavek na změnu – měníte funkčnost,
- pokud přidáváte novou funkčnost,
- pokud opravujete chybu,

[4] Viz http://jrefactory.sourceforge.net/

- ale také pokud se snažíte nalézt chybu – v tomto případě procházíte zdrojový kód, čtete ho a snažíte se nalézt příčinu, toto je vhodný okamžik pro přepis kódu do čitelné a lépe strukturované formy.

Úspěšnou podmínkou refaktoringu je existence vývojářských testů, které jsou kontrolním mechanismem pro předejití regresí (zavlečené chyby v případě změny zdrojového kódu). Při každém zásahu do kódu, při každé změně nás spuštění vývojářského (*unit*) testu objektivně informuje o dopadu dané změny. Bez vývojářských testů je refaktoring zdrojem problémů.

2.6.3 Párová práce

Nejznámější verzí párové práce je tzv. párové programování (*Pair programming*) velmi známé a propagované Kentem Beckem v Extrémním programování [Be99]. Párové programování vyžaduje dva spolupracující programátory u jednoho počítače. Oba se při práci vzájemně doplňují a střídají. Jeden může psát vývojářský test a druhý už přemýšlet o implementaci dané třídy. V průběhu párového programování se nemění pouze členové v dvojici u klávesnice, mění se také samotné páry – členové týmu rotují. Jedná se o plánovanou aktivitu, ne o ad hoc přístup podle potřeby (který je samozřejmě také přínosný). Párové programování má několik nezanedbatelných, jasně viditelných přínosů [Be99], [WK99]:

- Větší kvalita kódu – před kolegou programátorem tíhneme k tomu přijít s lepším řešením, než bychom použili, pokud bychom kód psali sami (psychotický aspekt, snaha vytáhnout se).
- Vícero vývojářů přispívá k návrhu aplikace – při rotaci párů se u kódu vystřídá více programátorů, jejich názory a řešení jsou konfrontovány s více lidmi, jejich znalost je přenášena na jiné.
- Jedná se o konstantí revize kódu – není třeba provádět další sezení, schůzky navíc, záběr je širší, než můžeme během formálních revizí kódu obsáhnout.
- Zvýšená morálka – pro některé je programování v páru zábavnější, než programování o samotě a pokud máme vedle sebe kolegu, tak opravdu programujeme, nesurfujeme na Internetu, nepíšeme e-maily kamarádům či manželce.
- Kolektivní vlastnictví kódu – pokud každý na projektu párově programuje a páry často rotují, každý člen týmu si vytvoří znalost zahrnující celou (většinovou) bázi kódu, nejen části IT služby či modulu (pokud to v daném kontextu jde). Každý je pak nahraditelný a dovolená, nemoc či odchod jednoho člena týmu neznamená pohromu pro určitou část IT služby.
- Mentoring – párové programování je nejjednodušší (rozuměj nejméně bolestná a také nejefektivnější) cesta předání a nabytí znalosti.
- Větší sounáležitost týmu – lidé se více poznají, střídají se ve spolupráci, naučí se spolu více spolupracovat a komunikovat.
- Programátoři jsou méně vyrušováni dotazy ostatních, jsou tedy méně často rozptylování od práce a nuceni přepínat kontext (tzv. *task switching*), který snižuje produktivitu.
- Potřebný menší počet pracovních stanic – extra (volné) stanice mohou být v průběhu párového programování využívány například pro proces

neustálého integrování (*Continuous Integration*) a testování či pro hledání dokumentace a možných řešení na fórech, v dokumentaci apod.

Technika párového programování má spoustu zastánců i odpůrců. ***Typickou námitkou manažerů*** je, že stejnou práci nyní dělají dva lidé, tudíž je produktivita těchto dvou lidí přibližně poloviční či že dostanou stejnou práci se zdvojenou kapacitou, resp. náklady (ano manažeři často vidí členy týmu jako zdroje či náklady). Další ***námitkou, ale naopak z řad vývojářů či pracovníků údržby*** způsobená pravděpodobně ješitností, jsou protesty proti párové práci, jelikož tvrdí, že nepotřebují nikoho do páru, že jsou dostatečně zkušení a dobří, že dělají stejně kvalitní řešení. Realita je však jiná, řada empirických statisticky významných studií a reportů dokazuje díky párovému programování mnohem kvalitnější řešení dodané v kratším čase, či oproti tradičním metodám a postupům vůbec dodané, viz například [Op07], [Con95], [Nos92], [WK99]. Jako jeden z příkladů uvedeme studii profesora Noska z Temple University provedenou v roce 1998 [Nos92]. Profesor Nosek studoval 15 zkušených programátorů řešících složitý problém po dobu 45 minut. Daný problém se vztahoval k jejich organizaci a programátoři ho řešili v jim známém prostředí, s jimi používanými nástroji. Zmíněných 15 programátorů Nosek rozdělil na 5 řešících problém samostatně (kontrolní skupina) a zbývajících 10 rozdělil do dvojic (experimentální skupina). Tyto dvojice pak řešili stejný problém párově. Jak jednotlivci, tak páry měli stejné podmínky, prostředí a k dispozici jim zmáné nástroje. Studie shrnuje výsledky pomocí statistických oboustranných T-testů. ***K překvapení všech manažerů i zúčastněných všechny experimentální skupiny (dvojice) překonali jednotlivce (kontrolní skupina)***. Dvojice také zmínili, že nebyly pod tlakem, ale řešení si více užívaly. Dvojice také více věřili svému řešení. Dvojice ***doručily řešení úkolu*** v průměru ***o 40% rychleji*** s tím, že doručili ***kvalitnější algoritmus***. Před experimentem byli programátoři skeptičtí ohledně řešení stejného problému ve dvojici, mysleli si, že to opravdu nemůže být zábavný proces. Výsledky studie však ukazují opak a to jak v efektivnosti a kvalitě výsledků, tak v jejich pocitu a požitku při společném řešení, který významně ovlivňuje motivaci jednotlivce [Nos92].

Jelikož některé z výše zmíněných výtek jsou na denním pořádku a zažité zvyky se překonávají jen těžce, je vhodné začít plánovaně s párovou prací (nejen programováním) malými kroky. Podle naší zkušenosti jsou nejlepší praktické výsledky, když nepoužíváme párovou práci po celý den všech 5 dní v týdnu, ale pouze v určitých plánovaných (ne tedy ad hoc) případech:

- Návrh implementace změny, kdy v páru navrhneme a probereme několik možných řešení podle pracnosti a dopadu – jedná se vlastně o instantní oponenturu návrhu. Samozřejmostí je pak také implementace vývojářského testu, který může napsat jeden vývojář, zatím co druhý již píše strukturu vybraného řešení.
- Revize kódu implementované změny/opravené chyby – samotné párové programování lze považovat za instantní revizi kódu. Přisedící s námi prochází to, co právě píšeme a může si všimnout nějaké chyby či nevhodné konstrukce. Navíc díky tomu, že říkáme a vysvětlujeme, co píšeme, neopomeneme nějakou základní aktivitu (typicky incializace proměnných, objektů či uvolnění zdrojů, testování objektů na *null*). Vyčleníme párovému programování tedy určitý časový úsek denně či

týdně podle velikosti změny/opravy (doporučujeme provádět hlavně s architektem softwaru či daného modulu).

- Spolupráce zkušeného a nezkušeného programátora – nezkušený se učí tím, že okoukává práci zkušeného a naopak v případě, že nezkušený sedí u klávesnice, zkušený na něj dohlíží, vysvětluje a instruuje ho. V případě potřeby se vymění, aby bylo vysvětlení prakticky ukázáno. Sdílení znalosti formou reálné práce je nejefektivnějším způsobem učení.
- Kontrola dodržování navržených mechanismů architektury (obecnější forma revize kódu) – architekt během dne párově programuje určitý čas (například 30 minut či 1 hodinu) s jednotlivými programátory, aby ukázal použití mechanismů architektury (u klávesnice architekt) a aby se ujistil, že jsou tyto mechanismy a pravidla následovány (u klávesnice programátor). Cílem je udržení kvalitního a přehledného kódu a pravidel architektury.
- Předání znalosti architektury softwaru – programátoři vývojového týmu předávají znalost architektury pracovníkům údržby pomocí párového programování (konkrétně přímo ukázkou opravení chyby/implementace změny) na pravidelné bazi – několikrát týdně v průběhu fáze předání znalosti či neustále v průběhu údržby (pokud se jedná například o týmy u stejné organizace).
- Párová práce jako mechanismus učení pro pracovníky na stejné úrovni – tento princip použijeme v případě, že oba pracovníci jsou na stejné úrovni znalostmi či schopnostmi. I přesto se mohou oba učit jeden od druhého. Každý má vždycky jiný kontext než druhý a každý programátor bude vždy mít i trochu jinou znalost základny kódu. Proto se vždycky může jeden od druhého naučit. Navíc díky sociologii, psychologii a fyziologii víme, že spojením kapacit dvou mozků vzniká vyšší než dvojnásobná tvůrčí síla, čehož můžeme také využít v tomto případě.

Zastáncům Extrémního programování (XP) se budou výše zmíněné názory zdát pravděpodobně málo extrémní. Z naší zkušenosti však tento mix přinesl nejlepší výkony a výsledky nejen pro naučení se této techniky a přináší je i v kontextu poměru investice/přínos. Správný poměr párové práce je pro každý projekt, tým a člověka jiný. Kritickým faktorem úspěchu párové práce je pravidelná obměna pracovníků pracujících v páru (například změna páru každý den či týden).

Většina z nás provádí párovou práci (návrh, programování či lokalizaci chyby) intuitivně, neplánovaně, ad hoc podle potřeby. Typickým příkladem takové párové práce v údržbě může být následující ukázka rozhovoru:

- ***Kolega:*** *„Mám tady někde chybu, kterou mám opravit a nemůžu ji lokalizovat, dělám na tom už asi 3 hodiny a pořád jsem nenašel příčinu. Nevíš čím to může být? Dělal si na tomto modulu minulý měsíc…“*
- ***Vy*** *(vyrušen v práci jdete pomoci kolegovi).*
- ***Kolega:*** *„Podívej, procházím log a zde to podle této výjimky způsobuje metoda této třídy. Když ji ale debuguji, tak … Ahá, díky moc za pomoc, to je ono!!!“*

- ***Vy (mírně podrážděně):*** *„No, nemáš za co…“ a jdete si sednout na své místo a vzpomínáte, kde byl ukončen váš myšlenkový tok.*

Toto rychlé odhalení příčiny chyby je přesně ukázka síly párové práce. Důslednost a postup krok po kroku, který zvolil kolega díky vaší přítomnosti pomohla odhalit zdroj chyby (a to pouhým následováním postupu, který by oba v páru následovali hned od začátku společného úkolu, ne až po 3 hodinách práce[5]). Výstupem takové ad hoc párové práce je sice úspěšné vyřešení problému, má ale několik neblahých důsledků. Prvním a také nejzásadnějším je neustálé vyrušování ostatních kolegů, což způsobuje přepínání kontextu a ztrátu jejich soustředění (tzv. *task switching*) vedoucí v nízkou produktivitu. Ochota takto vyrušovaných kolegů pomáhat je den ode dne menší. Druhým viditelným problémem je plýtvání času. Pracovník se ozval se svým problémem až po třech hodinách neúspěšného hledání zdroje chyby, v případě plánované párové práce mohl být problém vyřešen během několika minut. Úspěšnost neplánované párové práce také závisí na typologii daného člověka, extrovert se pravděpodobně nebude rozpakovat zeptat brzy, díky častému vyrušování ho ale kolegové mohou později ignorovat. Naopak introvert se nemusí zeptat vůbec a bude spoléhat na proaktivitu někoho ze zkušenějších kolegů.

Z těchto důvodů je plánovaná párová práce využívaná pro výše zmíněné účely mnohem efektivnější a přispívá také k lepší týmové práci a porozumění kolegům.

2.6.4 CI – Continous Integration

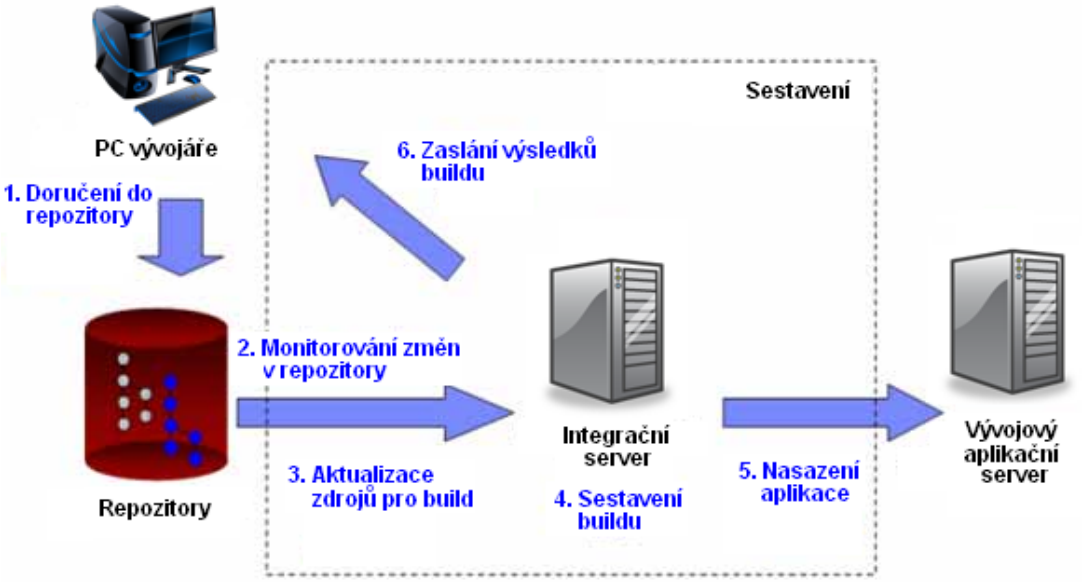
Neustálá integrace je sestavení komponent a jejich integrace do spustitelné verze aplikace nebo některé její části včetně případného spuštění určité sady testů. Tento proces je v případě iterativně inkrementálního vývoje a údržby důležitým prvkem. Spustitelná verze komponenty, sub-systému nebo celé aplikace slouží k neustálé demonstraci nových funkčností, odhalení integračních problémů, návrhových a programátorských chyb (rychlá zpětná vazba), ověření testy a také k demonstraci postupu na projektu. Každá verze je pod správou konfigurací, abychom ji mohli v případě potřeby použít. Pokud jsou naše iterace či intervaly sloužící pro opravu chyb krátké a tudíž potřebujeme sestavovat verze a testovat často, je automatizace tohoto procesu kritickým faktorem úspěchu. Důležité je také integrovat a testovat jednotlivé části vyvinuté samostatnými vývojáři nebo týmy. V této souvislosti se často zmiňuje technika či koncept zvaný neustálá, častá integrace (*Continuous Integration* – CI).

Martin Fowler a Matthew Foemmel [Fo06] definují neustálou integraci jako plně automatizované, opakované sestavování verze, které zahrnuje také testování a je spouštěno několikrát denně. To umožňuje vývojářům denně integrovat svůj nově napsaný zdrojový kód a předejít tak integračním

[5] Zde úřaduje tzv. nepozorná či nevědomá slepota, kdy jsme schopni přehlédnout viditelné chyby, změny či paradoxy, jelikož se naše soustředění zaměřuje pouze jedním směrem, viz například příspěvky a experimentální videa (http://psyche.cs.monash.edu.au/v5/psyche-5-03-mack.html, http://viscog.beckman.uiuc.edu/djs_lab/demos.html).

problémům či jejich dopad redukovat a hlavně získat rychlou zpětnou vazbu o kvalitě své práce. Na obr 2.9 je zřejmě viditelná síla této praktiky, její účinnost a následná nízká cena opravy chyby díky velmi rychlé zpětné vazbě a lokalizaci chyby.

Jednoduše lze říct, že neustálá integrace se skládá ze společného úložiště (tzv. *repozitory*) pro všechny členy týmu obsahující (nejen) aktuální kód a spustitelné soubory. Dále se skládá z automatického procesu sestavování verze a jejího testování. Tento proces může být spouštěn několikrát denně a je soběstačný, není třeba zásahu člověka. Je důležité říci, že ačkoliv neustálá integrace hodně závisí na nástrojích, není to jen použití nástrojů, ale spíše postoj k vývoji softwaru, který snižuje rizika plynoucí z pozdní integrace jednotlivých částí produktu. Blíže a více o neustálé integraci viz například článek na *Rational Edge*[6] či množství článků od Martina Fowlera[7].



Obr. 2-12: Postup a struktura neustálé integrace

Prostředí pro neustálé sestavení a testování by mělo být výsledkem procesu vývoje, stejně jako třeba sdílené úložiště zdrojových kódů (*repository*), konvence kódu, pravidla architektury nebo sada vývojářských (*unit*), funkčních a nefunkčních testů. Velmi často však tým údržby nemá tyto pravidla a informace k dispozici, proto je opět vhodné toto prostředí vytvořit krok po kroku.

V případě provozu a údržby rozsáhlého IS nebude mít smysl pravidelné sestavení celku, jelikož systém je již úspěšně provozován v produkčním prostředí a jasně odpovídá na otázku, zda jsme či nejsme schopni integrovat a spustit jej jako celek. Pokud tedy CI na úrovni služby neexistuje, nebude jeho vytvoření naším primárním cílem. Síla a význam CI v případě provozu a údržby bude především při kontrole změn a implementaci nových funkčností.

[6] http://www.ibm.com/developerworks/rational/library/sep05/lee/
[7] http://www.martinfowler.com/articles/continuousIntegration.html

Jelikož díky úspěšnému provozu víme, že je možné systém úspěšně sestavit, bude kritickou především každá změna rozhraní na externí systémy, kterých mohou být v praxi desítky. Smysl CI v provozu a údržbě je tedy především v pravidelném sestavení a pokrytí testy těch částí, které mají návaznost na externí systémy. Základem úspěchu a přínosu CI je komponentní architektura, resp. architektura s minimem závislostí. Typickou námitkou pro použití CI v kontextu sestavení a testování rozhraní na externí systémy je, že nemůžeme nasimulovat prostředí s desítkou dalších systémů, že je nemáme k dispozici. Tato námitka je relevantní, proto se pro simulaci chování externích systémů a testování jejich rozhraní používají objekty (zvané *mocks* a *stubs*) simulující základní chování či alespoň zasílání základní sady potvrzovacích a návratových dat externím systémem.

Postup vytvoření prostředí pro neustálou integraci komponenty obsahující rozhraní na externí systémy bude v údržbě opět velmi podobný jako v případě vývojářského testování a refaktoringu:

1. Jako první krok se pokuste vytvořit prostředí pro automatizované sestavení komponenty (předpokladem je komponentní architektura) z úložiště zdrojových kódů a její automatické nasazení na testovací server, čímž předejdete chybám generovaným člověkem v rutinním provozu, například zapomenuté kopírování určité změněné části, zdrojových souborů či konfigurací. Již samotné časté sestavení komponenty obsahující opravené chyby či změny chování vám odkryje možné integrační problémy.
2. Jako jeden z úkolů sestavení zařaďte také provádění základní sady testů (pro důležité či často měněné a opravované části) a také pro testování komunikace s externím systémem. Externí systém často za účelem testování nahrazujeme sadou objektů a dat, které imitují základní chování externího systému (použití tzv. *mock* a *stub* objektů).
3. Přidejte do prostředí a sestavovacího skriptu také konfigurační soubory, zdroje a další nezbytné soubory, které jsou součástí aplikace.
4. Při každé opravě chyb, změně kódu či implementaci nové funkčnosti přidejte do CI sady existující a také nové vývojářské testy.

Síla této praktiky je v automatizaci rutinní práce (předcházení lidským chybám), rychlé zpětné vazbě, lokalizaci chyb, ale také zkvalitnění návrhu a celkového řešení, což v konečném důsledku přispívá k celkové optimalizaci softwaru a vede k úspoře času a financí v jejím provozu a údržbě.

2.6.5 Retrospektiva

Retrospektiva (z latinského *retrospectare*, což znamená pohled zpět) vychází z tradičních projektů, kdy byla její jistá forma prováděna na konci každého projektu. Tým zde probral, co se na projektu osvědčilo, co nebylo vhodné vůbec a co například dělat jinak. Tyto poznatky byly užitečné a mohly být aplikovány na dalších projektech. Jejich největší přínos by však byl v rámci a kontextu daného projektu a týmu, jelikož ***zkušenost mezi projekty je přenositelná jen na výšší úrovni praktik a vzorů***. Na detailní úrovni lze dané poznatky použít jen a právě v tomto daném projektu. Pokud tedy děláme retrospektivu na konci projektu, je její přínos pro daný projekt nulový. Poznatky již nemůžeme použít k zefektivnění chodu současného projektu.

Proto jdou agilní přístupy dál i v této oblasti a retrospektiva probíhá mnohem častěji a několikrát v průběhu projektu, abychom se mohli neustále učit a zlepšovat naše chování a dosud vykonávané postupy. ***Retrospektiva v softwarovém projektu tedy probíhá na konci každé iterace***.

Základními otázkami, které bychom si měli v rámci retrospektivy klást jsou:

- Co fungovalo, co se osvědčilo v průběhu předchozí iterace projektu?
- Co moc dobře nefungovalo, co se neosvědčilo v průběhu předchozí iterace projektu?[8]
- Co bychom měli dělat jinak? Jaké konkrétní ***akce*** bychom měli ***do příští retrospektivy*** podniknout za účelem zlepšení projektu?

Cílem retrospektivy je vytvořit prostředí s vestavěným mechanismem kontroly, ponaučení a přizpůsobení (tzv. *inspect and adapt* přístup), což je v prostředí neustálých změn důležitým aspektem. Rozdíl oproti retrospektivě na konci projektu e možnost implementace akcí a také kontext a okamžitá zpětná vazba. Okamžité uvědomění si dané chyby (včerejší rozhodnutí bylo chybné) či naopak dobrého kroku umožňuje jeho předejití nebo naopak časté opakování a zlepšení současného projektu. Zásadním principem fungování a smyslu retrospektivy je generování akcí pro nápravu věcí, které nefungovaly, ale také pro opakování a dodržování věcí, které naopak fungovaly a měly pozitivní efekt. Pokud identifikujeme konkrétní problém, měla by následovat akce na jeho odstranění. Tato akce existuje ve formě konkrétního měřitelného úkolu pro osobu či tým, aby bylo možné zjistit (například při další retrospektivě) její provedení. Stejně tak postupujeme, pokud něco fungovalo. Pokud se vyplatil určitý krok, který běžně neděláme, je vhodné jej zařadit do našich pracovních postupů. Výsledkem pak může být aktualizovaná šablona, postup řešení problému či komunikace s uživateli, způsob testování, či doplněná knihovny nejlepších praktik týmu i organizace.

[8] Tuto část je nutné projít neosobně, nikoho za neúspěch nekritizovat ani nehodnotit (nejedná se o hodnocení osoby a jejích chyb, ale způsobu práce), ale naopak podívat se na to, proč se něco nepovedlo, proč a co můžeme udělat proto, aby se to příště nikomu již nestalo (pokud je to možné). Samozřejmě, vždy budou existovat určité výjimky, výskyty situací, které nemusí fungovat, nemusí se povést. Pro tyto výskyty je nesmyslné měnit způsob práce či přidávat nová pravidla.

Retrospektiva týden 6 (1. 2. – 5. 2. 2012)
Co fungovalo?

- [Honza] Předem naplánovaná párová práce s Mirkem sloužící pro hledání příčin problémů (během 1,5h odhaleny dvě příčiny a navržená možná řešení)
  → *Akce: naplánovat pro Mirka a Honzu v kalendáři párové hledání příčin pravidělně 2x týdně 1,5h. Ostatní mohou tento týden vyzkoušet podle domluvy.*
- [Zuzka] Pokusná aplikace vývojářských testů při dalším požadavku mi pomohla odhalit 4 chyby (přičemž 2 středně kritické), které bych jinak přehlédla.
  → *Akce: pokračovat v pilotování implementace vývojářských testů i pro oblast odstranění chyby (v následující iteraci).*

Co nefungovalo:

- [Tým] Poslední plánování iterace bylo neefektivní a trvalo příliš dlouho (4,5h oproti standardním 2h)
  → *Akce: Petr požádá mentora o facilitaci příštího plánování iterace a naplánuje dnes odpoledne schůzku, kde se pokusíme identifikovat příčinu.*

Co jsem se naučil tento týden, opakující se vzory, ostatní:

- [Honza] Nevytvářet si domněnky o druhých před jednáním – takový přístup nastaví myšlení a může zablokovat nalezení řešení či dosažení shody.

Součástí retrospektivy mohou být i oddíly, kdy diskutujeme jednu malou drobnou věc, kterou jsem se naučil během poslední iterace, resp. něco, co bych rád řekl, i když se může zdát, že je to pro ostatní zřejmé (realita taková vůbec být nemusí). Dále je možné do retrospektivy zařadit opakující se vzory, které jsme vypozorovali v našem způsobu práce, v našem způsobu přemýšlení, komunikaci se zákazníky. Prostě cokoliv, co by tým chtěl diskutovat a může ho to posunout blíže k efektivní komunikaci, k efektivnějším postupům.

Retrospektiva by neměla trvat příliš mnoho času, někdy může být dostačujících 10 minut (při pravidelných týdenních), při roční to naopak může být celý den. Dobrou praktikou je zapisovat vaše připomínky a pozorování do retrospektivy průběžně během týdne; pokud to necháte na daný okamžik, je možné, že si nevzpomenete na to důležité, co jste chtěli zmínit.

Důležitou podmínkou úspěšné retrospektivy je nehodnotit, nekritizovat druhé, ale objektivně se zaměřit na způsob práce, svoje chování a přínos. Je tedy nutné otevřené prostředí a kultura spolupráce v týmu. Bez spolupráce, otevřené komunikace, přijímání zpětné vazby a kritiky a bez sebereflexe je nemožné se zlepšovat a posunovat dále. Je však nutné podotknout, že právě retrospektivy toto prostředí mohou pomoci vybudovat. Pokud jsou vaše současné retrospektivy neefektivní, dlouhé, v atmosféře neklidu či strachu, a nepřináší

žádné zlepšení, požádejte o facilitaci zkušenou třetí osobu, mentora nebo kouče.

2.6.6 Denní schůzky

Denní schůzky (*Daily meetings*) jsou pravidelným tepem služby či projektu. Je na nich vyžadována účast každého člena týmu. Denní schůzky by se měly konat na stejném místě a ve stejný čas, jejich délka by pak neměla přesáhout 15 minut. Většinou se tato schůzka vede v jiné místnosti, než tým pracuje, a ve stoje, aby byla zachována co nejkratší doba a pozornost všech účastníků. Abychom mohli zachovat krátkou délku, je také třeba udržet na uzdě počet lidí, kteří se schůzky účastní. Efektivní počet je přibližně 7 +- 2 (překvapivě stejný počet jako efektivní velikost týmu). Koncept týmových krátkých denních schůzek zpopularizoval Ken Schwaber ve své knize o Scrumu [Sch04], i když známý byl tento koncept již předtím.

V průběhu denní schůzky každý člen týmu odpovídá následující otázky (a nic víc):

1. Co jsem udělal včera?
2. Co budu dělat dnes?
3. Co mně zdržuje v práci/brání v postupu?

Proč při vývoji a údržbě odpovídáme právě na tyto tři otázky? Mají svůj účel. První z nich zjišťuje zaměření týmu a také synchronizaci zúčastněných. Každý z týmu je tázán, zda nedělal na něčem, co nebylo plánováno pro danou iteraci a také zda opravdu dělal, co měl. Druhou otázkou revidujeme na denní bázi cíle projektu například z důvodu odhalených závislostí díky předchozí otázce. Poslední otázka může vygenerovat nové úkoly v našem seznamu úkolů. Nejdůležitějším cílem třetí otázky je identifikovat problémy bránící zdárnému dokončení projektu, které jsou poté přiřazeny manažerům, zákazníkovi či týmu k okamžitému řešení. Výsledkem je brzké odkytí problémů a jejich řešení již v zárodku.

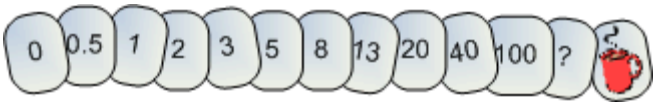
Je třeba zdůraznit, že denní schůzky neslouží k řešení problémů ani k reportování statusu manažerovi projektu. Nesprávné otázky můžou být následující: *„Kolik máme chyb? Kolik času jste strávili na svých úkolech? Kolik času jsme sdíleli znalosti? Kdo je pozadu podle plánu a kdo ne?“*. Taková situace ukazuje na mikro-řízení týmu. Denní schůzky tedy nejsou nástrojem byrokracie, ale nástrojem pro fungující a samozřejmou synchronizaci týmu. ***Pokud v rámci schůzky identifikujeme nějaký problém, řešíme ho až po jejím skončení*** s relevantními lidmi či řešitelskou schůzku naplánujeme na jiný čas, abychom neplýtvali časem ostatních členů týmu, kterých se daný problém netýká nebo nemají zájem se jeho řešení zúčastnit.

2.6.7 Odhadování v agilních projektech – Planning poker

Odhadování jako pravidelná týmová aktivita je jedna ze základních stavebních kostek iterativně inkrementálních přístupů. Cílem odhadů je znát přibližnou složitost scénářů a oprav chyb, které budeme implementovat v dané iteraci. V tradičně řízených projektech vypadá tvorba odhadů následovně. Odpovědná osoba (řešitel či manažer) vytvoří odhad sám, bez detailních znalostí

implementační stránky, bez konzultace s týmem, bez uvědomění si rizik. V nejhorším případě poskytne na přednesené požadavky přesné číslo (jedná se o odhad a on poskytuje přesné číslo jako závazek, navíc bez znalosti rizik a se spoustou nejasností!), které zákazník očekává nebo na ně má peníze. V lepším případě se odpovědná osoba zeptá týmu, jak dlouho bude daná implementace či oprava trvat. Typická odpověď v tomto případě může vypadat následovně. Někdo v týmu přemýšlí o ideálních člověko-dnech, někdo je mimo, jiného to nudí. První vykřikne nějakou hodnotu a ostatní přemýšlí, proč tolik či proč tak málo? Již jsou ovlivněni jeho názorem a řeknou jinou hodnotu, i když předtím chtěli třeba říct něco jiného.

Plánovací poker (pozor, nejedná se o známou hazardní hru) používá následující sadu karet:



Pokud se řešitel zeptá na odhady nyní, vybere každý člen týmu jednu kartu a schovanou ji nechá ležet. Když všichni vyberou kartu, současně ji otočí a ukáží její hodnotu. Rozpětí odhadu může být významné, takže členové krátce (max. 3 minuty) diskutují, proč si kdo vybral tak nízkou, resp. vysokou volbu. Nízká může znamenat, že vývojář opomenul důležité rozhraní a jeho implementaci či tvorbu dokumentace, testování či nezná danou oblast kódu detailně a usuzoval na základě mu známé části, která není pokryta testy. Po této krátké diskusi proběhne druhé kolo volby. Odhady týmu se většinou přiblíží jedné hodnotě (např. 3, 5, 5, 5, 8). V takovém případě se většinou vezme akeptovatelná hodnota pro všechny (5 v tomto případě) a pokračuje se odhadem dalšího scénáře.

Proč je číselná řada na kartách takto uspořádána?

- Cílem je zrychlení procesu odhadování limitovaným počtem čísel.
- Předejití falešného pocitu přesného odhadu (přesného závazku místo odhadu).
- Donutit tým rozbít komplexní scénáře/chyby do menších (je tedy jedno, jestli jeden zvolí 20, druhý 40 a třetí 100, není třeba se dohadovat, kdo je přesnější, toto jasně indikuje nutnost rozdělení scénáře/chyby do několika menších celků).

Jaký je význam speciálních karet bez čísla?

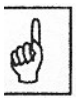
- Karta 0 znamená, že takový problém byl již řešen, scénář již implementován nebo má jasné řešení, resp. je daná práce zjevně jednoduchá (trvající jen pár minut).
- Karta otazník znamená, že daný člen týmu nemá absolutně žádnou představu o daném problému. Tato karta by měla být použita minimálně, jelikož její časté použití značí slabou znalost domény, potřeb zákazníka, jeho podnikových procesů. Na druhou stranu ale může odkrýt neznalost člena týmu, kterou můžeme začít řešit například párovou prací.
- Karta kafe znamená: *„Jsem příliš unavený, abych teď myslel, pojďme si dát krátkou pauzu.“*

Karty lze také použít pro efektivní hlasování v rámci týmu. Plánovací poker můžeme použít kdykoliv tým objeví nějaký problém či potřebuje rozhodnout zásadnější věc (pro operativní rozhodování tento postup postrádá smysl). Prvním krokem je odhalit a prodiskutovat všechny možné varianty řešení či postupu dosažení řešení. Pak jen tyto varianty očíslujeme a stejným způsobem jako je zmíněno výše volíme nám nejbližší variantu. Varianta s největším počtem hlasů může být brána jako výsledné řešení nebo může být podkladem pro další diskusi, podle závažnosti a odlišnosti jednotlivých řešení.

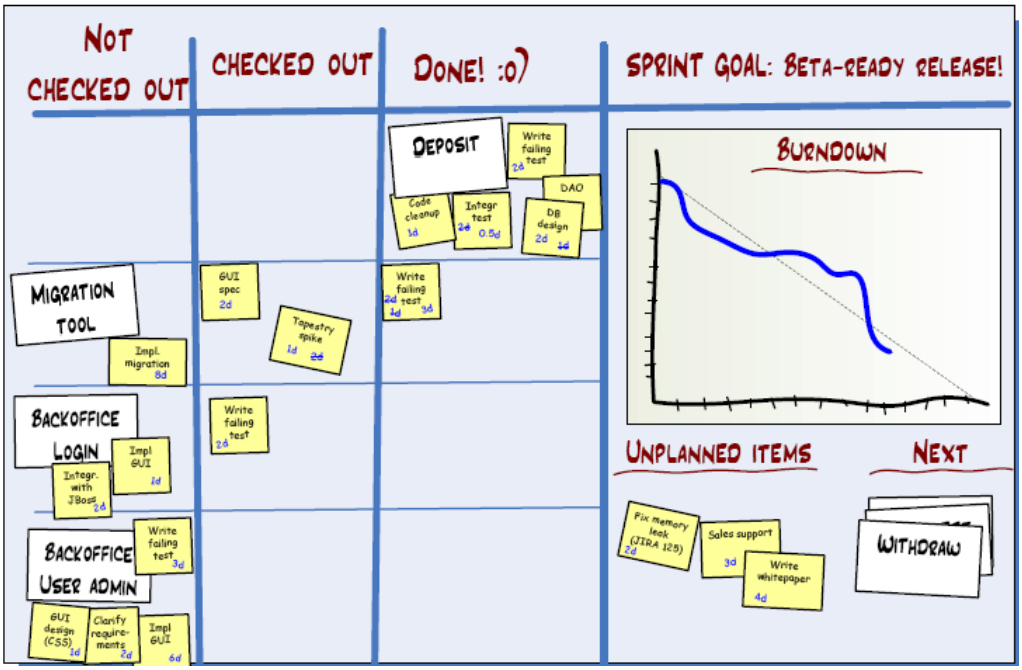
Zdrojem pro obrázky použité pro demonstraci plánovacího pokeru byl web společnosti Crisp, viz [Cr08].

2.6.8 Burn Down Chart

Nástroj *Burn down chart* vychází z populárního frameworku pro řízení projektů Scrum. Využívá několika základních principů pro monitorování postupu na projektu a odhalování problémů. Základem je rozdělení prací do maximálně denních úkolů (cca 2-8 hodin), menší jsou samozřejmě povoleny. Toto rozdělení provádí tým při plánování iterace. V průběhu iterace následujeme postup:

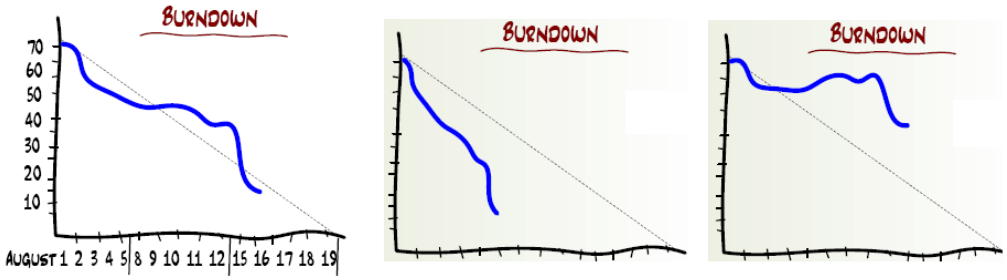


1. Na začátku iterace jsou všechny úkoly v kolonce „K implementaci“ či „Backlog iterace“ (na Obr. 2-13 sloupec *Not Checked out*). Počet těchto úkolů je pak vynesen na osu *y*.
2. Osa *x* značí dny iterace, na začátku iterace je den 0 a všechny plánované položky v backlogu iterace, na konci iterace (den D) by měly být všechny položky implementovány.
3. Jakmile začneme na některém z úkolů dělat, přesuneme ho do kolonky „Rozpracovaný“ (na Obr. 2-13 sloupec *Checked out*).
4. Pokud máme úkol hotový (např. 100% testů prošlo, integrace proběhla bez chyb a máme hotovou dokumentaci), přesuneme ho do kolonky „Hotovo“ (na Obr. 2-13 sloupec *Done*).
5. Scrum Master (SM) aktualizuje graf a vyvěsí jeho novou podobu.



Obr. 2-13: Burn down chart (zdroj: [Kn04])

Scrum Master případně manažer tedy denně aktualizuje graf podle vykonané práce a pokud je v průběhu grafu viditelná nějaká anomálie (graf neklesá nebo klesá příliš), manažer přijme nezbytné kroky. Ideální průběh grafu značí na Obr. 2-14 tečkovaná čára. První výskyt je realistický průběh grafem, někdy jsme s prací v iteraci mírně pozadu z důvodu špatného odhadu či skrytých prací, jindy zase o něco napřed. Ve výsledku jsme ale schopni cíle iterace naplnit a doručit v definovaném termínu, tj. na konci iterace. Pokud modrá linka značící aktuální průběh klesá příliš rychle oproti ideálnímu průběhu, je jasné, že jsme při plánování naše síly hodně podcenili a musíme do náplně iterace přidat další úkoly. To jsou ty konkrétní kroky, které Scrum Master nebo manažer přijme. Pokud modrá linka neklesá, okamžitě vidím problém a musíme málo prioritní položky z rozsahu iterace odebrat (tomu by samozřejmě měla předcházet diskuse se zákazníkem).



Obr. 2-14: Několik průběhů Burn down chartu (zdroj: [Kn04])

Tento denně aktualizovaný graf je vystaven na viditelném místě v místnosti, kde tým pracuje, aby měl každý člen týmu představu o průběhu iterace. Každý člen týmu také přistupuje ke svým úkolům a předělává je podle skutečnosti do správné kolonky.

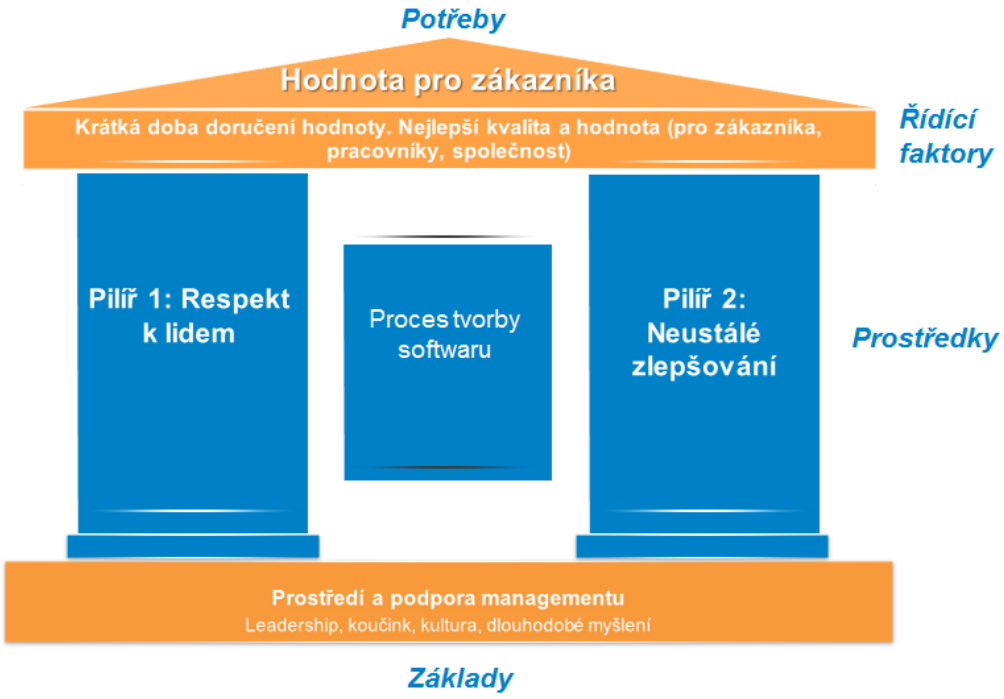
3 Lean development

Jedním z přístupů, o kterém se ještě musíme zmínit, je tzv. *Lean development*. Jedná se vlastně o aplikování zásad Lean manufacturingu[9] do softwarového inženýrství, abychom optimalizovali celý hodnotový řetězec v IT, od počátečního vývoje až po údržbu. Tyto principy pak poskytují praktické, měřitelné postupy k transformaci procesu vývoje a doručení software. Výbornou knihou pojednávající o tomto tématu je například [Po06].

Stejně jako Agile je i Lean hlavně způsobem myšlení a ne pouze nějakou konkrétní metodou a sadou technik nebo popisem procesu. Agile je v podstatě implementací Lean myšlení, ale Lean se nachází o jednu úroveň abstrakce výše. Základní principy Lean jsou následující:

- Dlouhodobé myšlení
- Leadership a koučink
- Respekt

Tyto základní principy se někdy popisují následujícím obrázkem domu. Základy, na kterých teprve můžeme něco stavět, jsou vhodné prostředí a správné zacházení s lidmi a pilíři, které umožní dodat hodnotu pro zákazníka je respekt k pracovníkům, zákazníkům i okolí a neustálé zlepšování činností, služeb a produktů.



Obr. 3-1: Základní principy a struktura Lean.

Dlouhodobé myšlení se podle Lean projevuje především tím, že veškerá rozhodnutí děláme směrem k dlouhodobému udržitelnému rozvoji a fungování společnosti. Pod pojmem společnosti myslíme opravdu celou společnost, nejen

[9] Způsob myšlení a řízení výroby a firmy známý například z Toyoty či jiných japonských firem.

vlastní firmu. Zajímá nás tedy dopad na okolí, zákazníky, naše zaměstnance i akcionáře. Nejsme řízeni výsledky v tomto měsíci nebo čtvrtletí na úkor dalšího vývoje, jako je tomu u tradičních firem. Typickou ukázkou je vývoj SW jako investice, ne jako náklad. Čím kvalitnější a lépe navržený systém k údržbě, tím levnější a méně problematická bude vlastní údržba, a i když vývoj samotný může stát 2x více, pořád se za léta provozu vyplatí. Toto je příklad dlouhodobého myšlení.

Leadership a koučink je hlavním rysem komunikace, podpory a řízení zaměstnanců. Neříkáme lidem, co mají dělat, ale snažíme se podpořit a směřovat jejich aktivitu směrem k vizi firmy. Např. místo příkazu, aby pracovník udělal to či ono se ho ptáme: *„jak bys tedy vyřešil tento problém, abychom zákazníkovi dodali nejlepší hodnotu a dodrželi naši vizi preferovaného dodavatele IT řešení pro velké firmy?“*. Já tuto situaci mnohdy hodnotím slovy: *„potřebujeme více skutečných a přirozených vůdců a méně manažerů“*, což je přesně jádrem leadershipu a koučinku. Tento přístup je těžké aplikovat ve společnosti, kde člověk egoisticky usiluje o manažerský post a pak přehlíživě zachází se svými podřízenými, nenaslouchá jim a nebo s nimi dokonce manipuluje. Koučing a leadership nemůžeme omezovat jen na úroveň manažerů, týká se všech úrovní společnosti. I pracovník může koučovat svého nadřízeného vhodnými dotazy.

Lidé jsou základem Lean organizace. Pouze šikovní lidé, kterým věříme a dáme prostor, mohou doručit opravdovou hodnotu zákazníkovi. Pracovníky bere vedení a management firmy jako sobě rovné, ne jako *„lidské zdroje“*, kterým se přikazuje. Respekt se projevuje také v tom, že se snažíme v době slabší poptávky pracovníky udržet a použít jejich znalosti a dovednosti na vylepšení služeb a produktů. Taková doba je totiž nejvhodnější k investici. Lidé mají čas, nejsou vytíženi projekty a zakázkami a tak může dojít ke skokovým inovacím. Tradiční firmy vidí v době nižší poptávky nevyužité zaměstnance jako finanční náklad a propouští je, což ukazuje nulový respekt k jejich bytí a hodnotě. Bohužel pro tyto tradiční firmy se zaměstnanci odchází i jejich jedinečná znalost a jméno takové firmy přitáhne jen druhořadé pracovníky, špička půjde jinam. Ani ztráta znalostí či jména není ve finančních výkazech vidět, tak to vlastně tradiční firmy netrápí, ale pak se tyto společnosti diví, že nejsou světové a tak dobré jako třeba Google nebo Facebook.

Principy, které jsme si nyní popsali, jsou ještě o jednu úroveň abstrakce výše než Lean principy. Proto, abychom pochopili, jak se tyto vysoce obecné principy projevují v praxi, musíme ukázat příklad praktik a nástrojů Lean.

3.1 Lean principy, praktiky a nástroje

Na stejné úrovni jako jsou Agilní principy definuje Lean principy vývoje, provozu a údržby softwaru. Jedná se vlastně o překlad původních principů tzv. štíhlé výroby (*Lean manufacturing*):

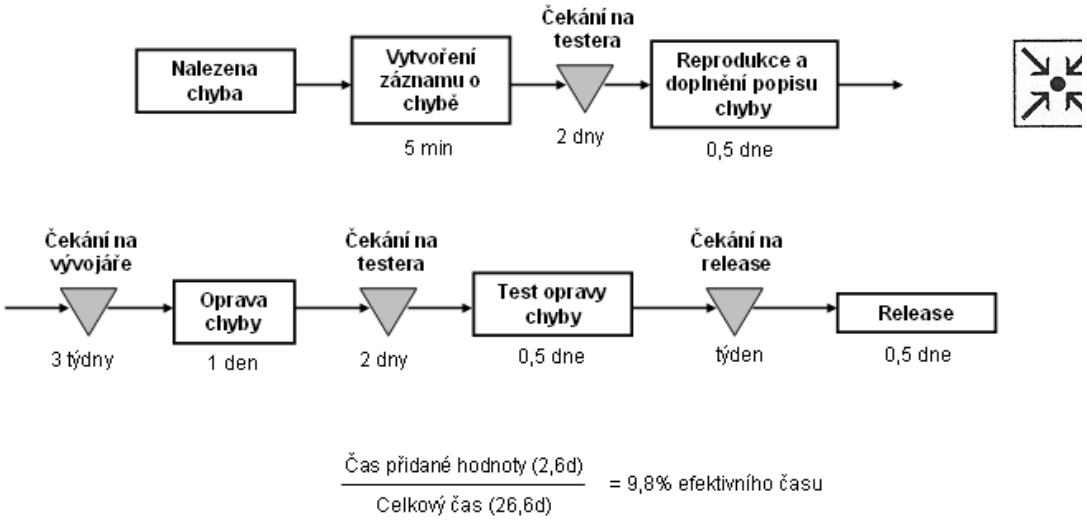
- ***Odstranění plýtvání*** (*Eliminate waste*) – základní princip zlepšování v Lean. V procesu vývoje softwaru vidíme každou aktivitu přímo nepřidávající hodnotu výslednému produktu jako plýtvání. Příkladem významného plýtvání při vývoji software jsou extra požadavky či

funkčně rozdělené týmy (vývoj a údržba jiným týmem, rozdílné a nekomunikující role), také předávání úkolu jinému člověku či týmu nebo čekání na schválení rozhodnutí je běžným plýtváním.

- ***Kvalita je součástí procesu/produktu*** (*Build in quality*) – princip zmiňuje kvalitu jako vestavěnou součást procesu. Pokud běžně odhalujeme problémy na konci díky verifikačnímu procesu, je náš proces pravděpodobně chybný. Stejně tak, pokud vývojáři provádí věci, které nechceme, aby dělali a naopak nedělají to, co chceme. Můžete porovnat s F principem RUPu, kdy je každý odpovědný za výslednou kvalitu produktu a vykonává své aktivity tak, aby byl doručen kvalitní produkt a ne, aby na konci procesu testeři začali odhalovat nekvalitní práci předchozích rolí.
- ***Vytváření znalosti*** (*Create knowledge*) – důležité pro úspěch projektu je pochopení potřeb zákazníka. Zákazník se v průběhu projektu učí co chce, stejně jako my, je tedy vhodné poskytnout strategii (např. iterativní vývoj) podporující odhalení potřeb zákazníka a následné okamžité využití těchto poznatků. Pro tento účel je také vhodné mít připravený soubor znovupoužitelných standardů a průvodců, které mohou lidé jednoduše modifikovat a použít pro potřeby specifického projektu. Konzistentní a včasná zpětná vazba pro tým je také důležitá a napomáhá učení a vytváření znalosti.
- ***Odložení rozhodnutí*** (*Defer commitment*) – není důležité začínat vývoj software kompletní definicí požadavků. Místo toho je efektivnější podporovat byznys flexibilní architekturou, která je tolerantní ke změnám tím, že odložíme nezvratná rozhodnutí do pozdějších fází, do nejzazšího možného momentu. Pro tento způsob přijímání rozhodnutí je nutné mít těsnou spolupráci s byznysem a znát možné byznys scénáře, tj. jakým směrem se může ubírat byznys zákazníka: růst počtu zákazníků, nové produkty, trhy, ústup z pozic, zachování atd.
- ***Brzké doručení*** (*Deliver quickly*) – rychlé doručení kvalitního systému můžeme dosáhnout tím, že nevyžadujeme po týmu víc, než je díky své omezené kapacitě schopný vykonat. Místo toho necháme na týmu ať se řídí sám tak, aby doručil maximum kvalitní práce a ptáme se jen co je/není tým schopen v daném čase dokončit a doručit.
- ***Optimalizace celku*** (*Optimize the whole*) – předtím, než začneme optimalizovat, je třeba znát celý kontext, celý obraz situace. Je třeba znát podnikové procesy, které má vyvíjený systém podporovat či již podporuje, je třeba vzít v potaz všechny zainteresované týmy, všechny zúčastněné strany, abychom zákazníkovi doručili kompletní produkt. Místní optimalizací, bez znalosti celého řetězce, můžeme snadno dosáhnout zhoršení výkonnosti celého procesu či vykonávání zbytečných kroků.

Agilní praktiky a techniky pomáhají zavést Agilní principy do denního života týmů a manažerů. Stejně tak Lean nástroje pomáhají úvést tyto myšlenky do praxe. Pro pochopení si uvedem některé z Lean nástrojů:

- ***Mapa hodnotového toku*** (tzv. *Value Stream Map*) – využívá síly vizualizace a slouží k popisu procesu, jakým způsobem doručujeme hodnotu zákazníkovi, tj. jaký je sled aktivit, kde se čeká, kde je úzké místo – viz následující příklad:



Z této mapy je patrné, kde se vyskytuje největší plýtvání. Nejedná se o vlastní aktivity, nýbrž o všechna ta schvalování, čekání, předávání. Pokud tedy chceme mít efektivnější proces, musíme se zaměřit na odstranění těchto věcí. Smyslem tvorby VSM není mít detailně přesný faktický proces, ale spíše dostatečně kvalitní vizualizaci současného stavu spolu s čísly z našich nástrojů.

- ***5x proč*** – technika pro zjištění příčin problémů. V osobním životě i v práci jsme zvyklí reagovat na situace, které nastanou, ale ty jsou většinou jen symptomem různých hlubokých problémů, které nejsou viditelné a řešené. Bolí nás hlava proto, že málo pijeme, my ale symptom zaženeme práškem. Máme chybový software, protože slibujeme nemožné a řešíme to více přesčasy a dalšími lidmi na projektu, což situaci jen zhoršuje apod. Systémové myšlení nám říká, že pro odstranění těchto symptomů musíme vyřešit kořenovou příčinu a ne samotné symptomy. Jakmile vyřešíme příčinu, symptomy zmizí samy.

Problém (symptom): Přišli jste do garáže a všimli jste si píchnutého kola
Proč? Protože jsou na podlaze rozsypané hřebíky
Proč? Protože je kartonová krabice v polici mokrá
Proč? Protože prosakuje střechou

Symptomem na uvedeném příkladu je píchlé kolo, vyřešit jej můžeme výměnou kola nebo uklizením rozsypaných hřebíků. Kolo ale můžeme v garáži píchnout opakovaně. Kořenovou příčinou je díra ve střeše, kterou musíme opravit. Tudíž věc, kterou bychom si na první pohled asi s píchlým kolem nespojili.

Kontrolní otázky:

1. Co je to agilní manifest?
2. Na jakých hodnotách staví XP?
3. Co je to refaktoring?
4. Jaký je rozdíl mezi sprintem a iterací?
5. Vyjmenujte tři principy Lean Developmentu.
6. Jmenujte jeden Lean nástroj a jeho účel.

Úkoly k zamyšlení:

Párové programování je velmi zajímavou a efektivní technikou. V našich končinách a pro některé lidi však není vhodné po celých 8 hodin denně, 5 dní v týdnu. Zamyslete se nad tím, jak lze párové programování variovat, zda je nutné je provádět vždy po celou dobu pracovní doby a zda tyto obměny mohou přinést či ztratit hodnotu.

Korespondenční úkol:

Zmínili jsme, že v iterativním způsobu vývoje netvoříme na začátku detailní plán celého projektu, ale pouze jakousi road map. Pokuste se zamyslet nad obsahem takového dokumentu celkového plánu projektu, jelikož nějaký celkový plán je určitě nutné vytvořit. Co (časy, zdroje, milníky, ...) by podle Vás měl takový plán obsahovat a proč?

Shrnutí obsahu kapitoly

V této kapitole jsme detailněji představili agilní přístupy a jejich techniky, které už byly zmíněny a propagovány v textu Informační systémy 1. Představeny byly nejznámější přístupy Scrum, XP a Crystal a FDD stejně jako denní mítinky, refaktoring, neustálá integrace a TDD či retrospektivy. Dále jsme představili Lean Development, jako příklad systémového myšlení.

4 Provoz podpora a údržba

V této kapitole se dozvíte:

- Proč je nutné definovat procesy pro údržbu a podporu provozované aplikace?
- Definice základních pojmů provozu, podpory a údržby.

Po jejím prostudování byste měli být schopni:

- Pochopit potřebu existence standardních procesů pro provoz a údržbu.
- Vyjmenovat procesy ITIL a jejich účel.
- Popsat základní koncepty provozu a údržby

Klíčová slova této kapitoly:

Procesní řízení, provoz a údržba, ITIL.

Doba potřebná ke studiu: 2 hodiny

Průvodce studiem

Kapitola představuje oblast provozu a údržby vyvinutého software/informačních systémů pomocí standardních postupů, konkrétně pomocí procesů definovaných frameworkem pro provoz a údržbu ITIL. Text vysvětluje, co je to ITIL a na příkladu ukazuje jeho aplikaci.
Na studium této části si vyhraďte 2 hodiny.

Dosud jsme se v tomto textu mimo jiné zabývali iterativním vývojem software, který je zákazníkovi doručován v pravidelných releasech. Na základě zpětné vazby je pak možné aplikaci dále upravovat, doplňovat a vylepšovat, aby zákazník dostal řešení, které skutečně řeší jeho problémy v business doméně, byl spokojený a toto řešení mu přinášelo zisk a konkurenční výhodu. Stejně důležitou částí jako vývoj softwarového produktu je však také jeho provoz a údržba. Právě na tuto oblast se zaměříme v následujícím textu.

Pojem podpora či obecně údržba (anglicky: support, resp. maintenance) je všeobecně znám a intuitivně by ho asi dokázal popsat každý. IEEE definuje tuto disciplínu následovně [IEEE]:

> "*Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment.*"

Česky bychom výše zmíněnou definici mohli volně přeložit následovně:

> *„Údržba softwaru je procesem modifikace softwarového systému či komponenty po doručení uživateli za účelem korekce chyby, zlepšení výkonu či jiného atributu nebo adaptace systému změnám prostředí.“*

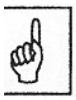
Tato definice obsahuje jednu diskutabilní myšlenku a tou je počátek disciplíny podpory a údržby začínající až v momentě doručení softwaru uživatelům. IT experti tuto problematiku diskutovali a výsledkem jejich diskusí byl procesně

orientovaný ISO model životního cyklu vývoje software. Tento životní cyklus zahrnuje disciplínu údržby software, která může začínat již brzy v rámci životního cyklu projektu a zabývá se opravou chyb.

Výše zmíněné standardy či doporučení také definují tři kategorie údržby software. Jedná se o následující [IEEE], [ISO1]:

- Korekce, opravy (*Corrective maintenance*) – reaktivní modifikace softwarového produktu za účelem korekce objeveného problému, chyby, které jsou vykonávány po doručení produktu do produkce/zákazníkovi.
- Přizpůsobení (*Adaptive maintenance*) – modifikace softwarového produktu za účelem změny funkčnosti z důvodu měnícího se prostředí. Tyto modifikace jsou také vykonávány po doručení produktu do produkce/zákazníkovi.
- Zdokonalení (*Perfective maintenance*) – modifikace softwarového produktu za účelem zlepšení výkonnosti (tzv. *performance*) či jednoduchosti údržby (tzv. *maintainability*) prováděné taktéž po doručení produktu do produkce/zákazníkovi.

Obecně v IT a také dále v textu se můžeme setkat s pojmem podpora informačního systému (anglicky *support*). Tento pojem je dokonce dobře známý i ne IT uživatelům[10], což je u jiných pojmů z naší exotické IT řeči spíše výjimka. Podívejme se tedy opět, jak IEEE definuje tuto disciplínu [IEEE]:



> *„Product support is providing of information, assistance, and training to install and make software operational in its intended environment and to distribute improved capabilities to users.“*

Česky bychom zmíněnou definici mohli volně přeložit následovně:

> *„Podporou produktu rozumíme poskytování informací, pomoci a tréninku za účelem instalace či provozu software v prostředí k tomu určeném a také jí rozumíme distribuci zlepšených funkčností software ke koncovým uživatelům.“*

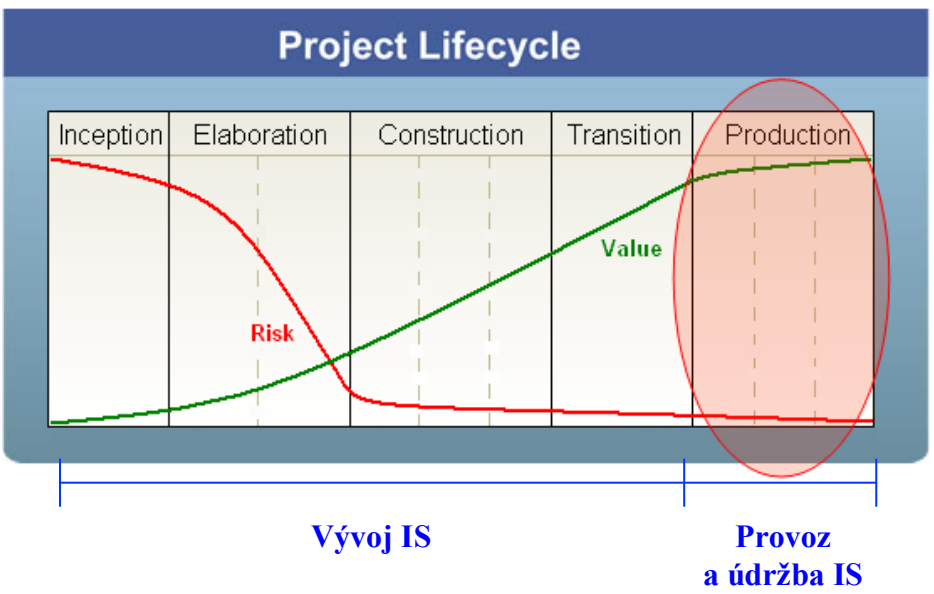
Z výše zmíněných definic a také druhů údržby je tedy zřejmé, že pojmy podpora a údržba se týkají aktivit spojených s provozem informačního systému, korekcí chyb, zlepšením výkonu či úpravy funkčností, ale také aktivit spojených s poskytováním informací a tréninku, ***vše převážně po konečném dodání systému zákazníkovi***. Důležité je však upozornit na slovíčko převážně. Běžně totiž existují činnosti, které provádíme (a často i musíme) již před samotným nasazením u zákazníka. Mezi příklad takových aktivit patří následující:

- Nábor pracovníků na pozice v podpoře.
- Školení pracovníků podpory (např. formou práce v páru na reálných úkolech).

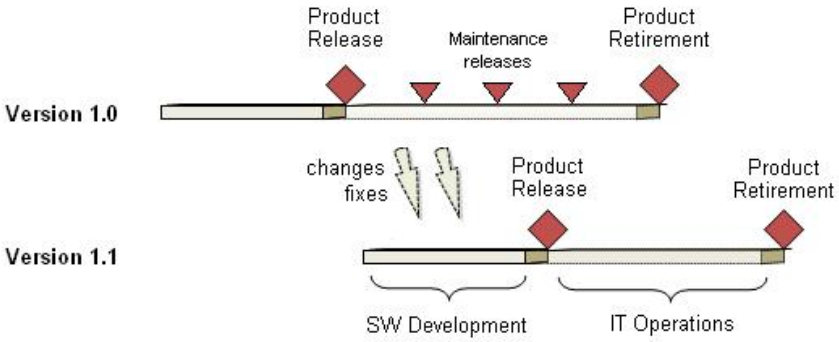
[10] Pravdepodobně téměř všichni z nás znají různé vtipy a opravdové situace zaznamenané pracovníky podpory (příklad blondýny s myší v pravém horním rohu pokoje; „technických expertů“ hlásících problém s nezapojenou tiskárnou atd.)

- Sdílení, předání a dokumentace znalostí od dodavatele systému k provozovateli poskytujícímu podporu (nejlepším řešením je využití členů vývojového týmu i pro podporu a provoz, jelikož tito lidé znají systém nejlépe).
- Příprava dokumentace a postupů, procesů (toto může být například forma učení se systému).
- Příprava prostředí – instalace serverů a sítí, operačních systémů, databází.

Následující obrázek (Obr. 4-1) ukazuje místo provozu a údržby v životním cyklu informačního systému. Provoz a údržba tedy oficiálně začíná dnem nasazení aplikace do ostrého provozu, i když už víme, že v realitě musíme provádět určité akce (školení personálu service desku, přenos znalostí, příprava prostředí, pilotní provoz) dřív, než tato fáze oficiálně začne. Obr. 4-2 pak ukazuje návaznost mezi jednotlivými verzemi softwarového systému a jejich nejčastější vstupy.



Obr. 4-1: Životní cyklus projektu a místo provozu a údržby v jeho kontextu. Obrázek je trochu zavádějící, neproporcionální, jelikož provoz a údržba trvají velmi často několikrát déle (řádově i desítky let) než vlastní vývoj informačního systému (měsíce až roky).



Obr. 4-2: Rozdílné verze systému a jejich životní cyklus a vstupy (změny, opravy chyb)

Jako poslední pojmy v této kapitole si vysvětlíme pojem provoz ve významu anglického *IT Operations Management* a pojem správa aplikací ve smyslu anglického *Application Management*.

IT Operations Management (OM) je podle [IEEE]:

> *„Proces provozu počítačového systému v určeném prostředí pro určené účely."*

ITIL [ITIL] popisuje provoz jako funkci odpovědnou za neustálou správu a údržbu IT infrastruktury organizace s cílem zajistit dohodnutou úroveň IT služeb pro byznys. Jedná se tedy o sadu denodenních aktivit prováděných za účelem provozu IT infrastruktury. Mezi prováděné aktivity patří:

- Správa a řízení provozních aktivit a událostí v IT infrastruktuře.
- Správa, provoz a monitoring pracovních stanic, serverů a datových center.
- Zálohy a obnovy dat a nastavení.
- Plánování a provádění rutinních úloh, skriptů.
- Správa a řízení tiskových a jiných výstupů.
- Fyzická a logiská správa a konsolidace serverů a datových úložišť.

Pojem správa aplikací ve smyslu anglického *Application Management* (AM) je pak literaturou vysvětlován následovně. AM je odpovědný za správu aplikace v průběhu životního cyklu a hraje také významnou roli při návrhu, testování či zlepšování aplikačních částí IT služeb. AM tým bývá také často součástí vývojových projektů, ale nejedná se o vývojový tým. AM pomáhá s rozhodováním, zda informační systém nakoupíme, vytvoříme vlastními silami, jak bude probíhat integrace, vše s cílem podpořit byznys procesy organizace. Dovednosti AM týmu pokrývají jak disciplíny specifikaci požadavků, pomoc při návrhu, případně vývoji, tak také disciplíny testování, integrace a nasazení a následnou údržbu a rozšíření aplikačního softwaru či informačního systému.

Pro bližší pochopení rozdílů těchto dvou světů si ilustrujme na dvou příkladech pohled a jazyk člověka z jedné a druhé strany:

Pojem	Pohled AM	Pohled OM
Změna	Změna funkčnosti, chování softwarového systému	Změna čehokoliv v IT infrastruktuře (tj. základní SW, databáze, HW či také nárůst kapacity úložiště, vyladění výkonnosti)
Konfigurace	Všechny části softwarového systému / IT služby v daných verzích sloužící k běhu systému / služby	Cokoliv v IT infrastruktuře

Tabulka 4-1: Rozdíl a zaměření správy aplikací a správy infrastruktury

Pozorný čtenář si jistě všimne, že v principu se jedná o stejné pohledy, jen je předmětem jiná věc: aplikace vs. celé IT z pohledu hardware, sítí a základního software. V podstatě by se tento pohled dal zjednodušit na hmatatelné (infrastruktura) vs. nehmatatelné (aplikace, software, informační systém). Tímto se dostáváme k problematice existence různých skupin lidí v organizaci a jejich světů a z nich plynoucích jiných jazyků.

Kritickým faktorem pro úspěšný vývoj, doručení a provoz softwarového produktu je kooperace tří skupin: jedná se o zákazníky a uživatele, dále o vývojáře a v neposlední řadě o provozovatele (údržbu) aplikace.

Následující příklad se snaží ukázat problémy, které nás mohou potkat při provozu existujícího software. Dozvíme se, proč je důležité mít definované standardní procesy a jaké v této oblasti existují ověřené postupy a praktiky.

4.1 Špatný scénář

Provoz aplikace doprovází několik nutných a pravidelných zásahů. Pokud provozujeme ekonomickou aplikaci, je třeba dbát na to, abychom implementovali nové zákony a směrnice. Pokud provozujeme výrobní aplikaci, je třeba reflektovat změny technologie, výrobních linek apod. Evidence, ohodnocení, implementace, testování a integrace změny (či nové funkčnosti) je však pouze jedním zásahem do provozované aplikace. Daleko závažnějším případem, který je třeba co nejdříve řešit, je výpadek aplikace nebo neočekávané chování, které znemožňuje uživatelům systému práci. Čím déle a čím více uživatelů nemůže pracovat, tím více peněz to organizaci stojí.

Jednoduchý příklad:

Náklady na jednoho zaměstnance	.....	500 Kč / h
Počet zaměstnanců	.....	200 celkem
Výpadek aplikace	.....	3h

Nemožnost práce 30 zaměstnanců po dobu 3 hodin z důvodu výpadku aplikace stojí organizaci: 30 x 3 x 500 = 45.000 Kč!!!

Z tohoto důvodu potřebujeme mít také mechanismy, které nás o výpadku informují, a nástroje, které nám pomohou výpadek vyřešit. S tím souvisí nejen hledání příčiny, ale také hledání informací o hardware a software daného uživatele, o jeho konfiguraci a jednotlivých verzích software, který používá.

Dobrým pomocníkem je v tomto případě také znalostní báze obsahující řešené problémy s popisem příznaků a s řešením.

Následující příklad ukáže, jaké problémy může způsobit neexistence těchto mechanismů (procesů). Budeme se zabývat řešením incidentů, problémů (skrytá příčina způsobující incidenty) a změn od počátku (jejich nahlášení či zjištění) až po jejich vyřešení (implementace do provozního prostředí). V příkladu si ukážeme špatný i dobrý scénář, aby byl čtenáři patrný rozdíl.

Účastníci:
Mary … uživatel mající problém,
Pete … aplikační programátor,
John … systémový administrátor,
A další osoby vystupující podle potřeby.

Pokuste se sami najít problémy a jejich příčiny v následujícím „katastrofálním“ případu. Některé kroky jsou naznačeny cíleně příliš extrémně, aby byly možné problémy viditelné na první pohled, ale scénář samotný vychází z autorovy zkušenosti a zvýrazněné jsou jen drobné části.

Pondělí ráno

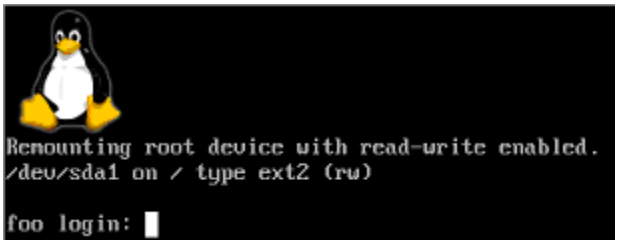
- Mary používá ke své práci program SkladAObjednávky v1.1, po hodině práce program přestane odpovídat, zhroutí se a již nejde znovu spustit.
- Mary zavolá aplikačního programátora Peta od dodavatele, jelikož jsou kamarádi a znají se, aby jí pomohl s daným problémem.
- Okamžitě po hovoru na to Pete zapomene, jelikož má spoustu práce s chystaným buildem. Mary se však v průběhu společného oběda připomene.
- Pete se jí při obědě zeptá na nějaké symptomy (chybové hlášky, apod.), Mary si už na moc od rána nevzpomíná, ale něco Petovi přece jen poví.
- Po obědě se však Pete opět vrací ke své práci a na Mary zapomíná.
- Mary pořád nemůže pracovat s aplikací na zpracování objednávek a dělá nedůležitou práci.

Pondělí odpoledne

- Mary opět volá Petovi, aby se informovala o pokroku.
- Pete se naštve, protože ho Mary vyrušuje od práce a on potřebuje dokončit build pro nasazení u zákazníka, které má proběhnout následující dny.
- Pete tedy přestane programovat a začne se zabývat identifikací problému, proč aplikace nefunguje jak má.
- Mary pořád vykonává nepodstatnou práci, důležité objednávky stále nejsou zpracovány.
- Odpoledne v 15h jde Mary domů, protože tuší, že aplikace stále nejede.
- Pete zůstává v práci až do 20h a hledá kde je program provozován, na jakých serverech běží (HW) a jaký middleware – aplikační server a databázi (SW) – systém používá.

Úterý ráno

- Pete potřebuje dokončit build určený k nasazení, proto přichází dřív do práce i přesto, že včera pracoval do večera.
- Mary přišla do práce raději později než obvykle, protože si nebyla jista, zda už bude její aplikace funkční.
- Poté, co Pete dokončil build, pokračuje v hledání serverů z předchozího dne – nachází je, jedná se o Linux servery.
- Pete je naštěstí linuxový nadšenec a tudíž zná Linux prostředí (příkazovou řádku), ale bohužel nemá uživatelský účet.
- Pete se ptá Johna (sysadmin) na nějaké přihlášení.
- John jako Petův dobrý kamarád mu dá root heslo, protože počítá, že si vytvoří účet a nahraje tam nějaké nové mp3 a filmy, proč jinak by se na tento server ptal...



Úterý odpoledne

- Pete však nemá čas řešit tvorbu svého účtu, přihlásí se tedy jako root a najde program SkladAObjednávky v1.1.
- Náhodou si při startu MC všimne, že je disk serveru plný.
- Smaže tedy nějaké dočasné a log soubory, to vše jako root.
- Poprosí Johna aby restartoval Oracle DB a Apache Tomcat, jelikož zjistil, že je program využívá a že oba nejedou.
- John je restartuje, bez toho aniž by někdo uvědomil další uživatele serveru. Nyní program SkladAObjednávky v1.1 opět běží.
- Pete volá Mary, že vše funguje a Mary začíná opět pracovat s aplikací.
- Pete se vrátí k práci na buildu pro zítřejší nasazení – spouští a analyzuje testy.
- Zůstává v práci opět až do 20h do večera, aby vše odladil a nachystal.

Středa ráno

- Mary zpracovává poslední čekající objednávky, ale po 2 hodinách práce se opět objeví stejný problém.
- Volá Petovi, ale nikdo telefon v jeho kanceláři nebere, jelikož je Pete u zákazníka a instaluje build, který poslední dva dny dolaďoval a testoval.
- Mary mu tedy zavolá na soukromý mobilní telefon, jelikož má jeho číslo.
- Pete volá zpátky Johnovi, aby se na problém podíval, mohl by s ním od včerejška alespoň trochu obeznámen – ví o které servery se jedná.
- Až nyní Johnovi dochází, co na serveru včera Pete dělal. John opět vidí plný disk, zálohuje na jiný stoj a smaže původní dočasné soubory a některé nepotřebné logy.
- Poté restartuje opět všechny služby aplikačního serveru a databázi a vidí, že opět všechno funguje správně.

Středa odpoledne

- John píše skript, který bude pravidelně zálohovat na jiný server a mazat dočasné soubory a některé nepodstatné logy.
- John si také začal kopírovat všechny logy programů běžících na tomto serveru, aby mohl zjistit, kde je chyba, proč se disk stále plní.
- Mezi tím, než se mu nahrají všechny soubory začne analyzovat již stažené, asi po hodině práci si všimne, že se velmi dlouho stahuje Oracle DB log – důvodem je jeho velikost 25 GB!!!
- Prozkoumá tedy tento log přímo na serveru a zjistí, že obsahuje programátorské výpisy.
- Přepíše skript tak, aby mazal hlavně Oracle log a původní zálohuje.
- Poté informuje Mary, že může opět pracovat s aplikací a problém se již neopakuje.
- John prohledává Internet a hledá ve fórech, zda se s tímto problémem již někdo setkal a zda je problém řešen. Nenajde žádnou odpověď, reportuje tedy chybu společnosti Oracle.

Závěry:
I když je daný příklad velmi ostrý a chyby viditelné, spousta organizací opravdu pracuje na podobných principech a nepřipadá jim to divné, jsou-li na toto upozorněny. To je bohužel také autorova zkušenost z jedné firmy. Z tohoto příkladu můžeme udělat několik závěrů:

- Pete je přetížený a naštvaný.
- Build, na kterém pracoval, může obsahovat chyby, kterých si díky únavě a napětí nemusel všimnout.
- Pete přistupuje na servery, kam nemá právo přístupu a to dokonce jako root a navíc zde maže jako root soubory!
- Mary nemohla zpracovávat téměř dva dny objednávky, které generují zisk a mohly vést až k zastavení firmy!
- Znovu se opakující incidenty.
- Kořenová příčina incidentů stále neřešena!
- John jako administrátor začíná zkoumat problém (dělat svou práci) až třetí den od incidentu.

4.2 Lepší scénář podle ITIL

Nyní si povíme, jak by stejný případ mohl vypadat v případě, kdyby daná organizace měla implementovány procesy podle ITIL. Opět se zde vyskytují stejní účastníci, jen průběh řešení incidentu je odlišný.

Účastníci opět stejní:
Mary … uživatel mající problém,
Adam … pracovník podpory,
A další osoby vystupující podle potřeby.

Pondělí (Incident Management):

- Mary používá ke své práci program SkladAObjednávky v1.1, po hodině práce program přestane odpovídat a zhroutí se a již nejde znovu spustit.

- Mary vytvoří záznam o incidentu[11] v Service Desk nástroji, který je jediným kontaktním místem (SPOC – *Single Point Of Contact*). Záznam je reportovaný na IT službu SkladAObjednávky.
- Incident manažer přiřadí kategorii (aplikace) a prioritu (vysoká) incidentu a přiřadí incident Adamovi.
- Mary je o přiřazení incidentu řešiteli notifikována automaticky e-mailem.
- Adam obdrží o svém přiřazení taktéž notifikaci, přeruší aktuální práci z důvodu vysoké priority nového incidentu a začne jej řešit.
- Adam začne prozkoumávat incident s použitím znalostní báze (KB – *Knowledge Base*), konfigurační databáze CMDB a automatických nástrojů.
- Adam ví, na kterém serveru program běží a jaké jiné programy využívá díky katalogu IT služeb a informaci o relevantních konfiguračních položkách této služby (viz Tabulka 4-2, sloupec vpravo).

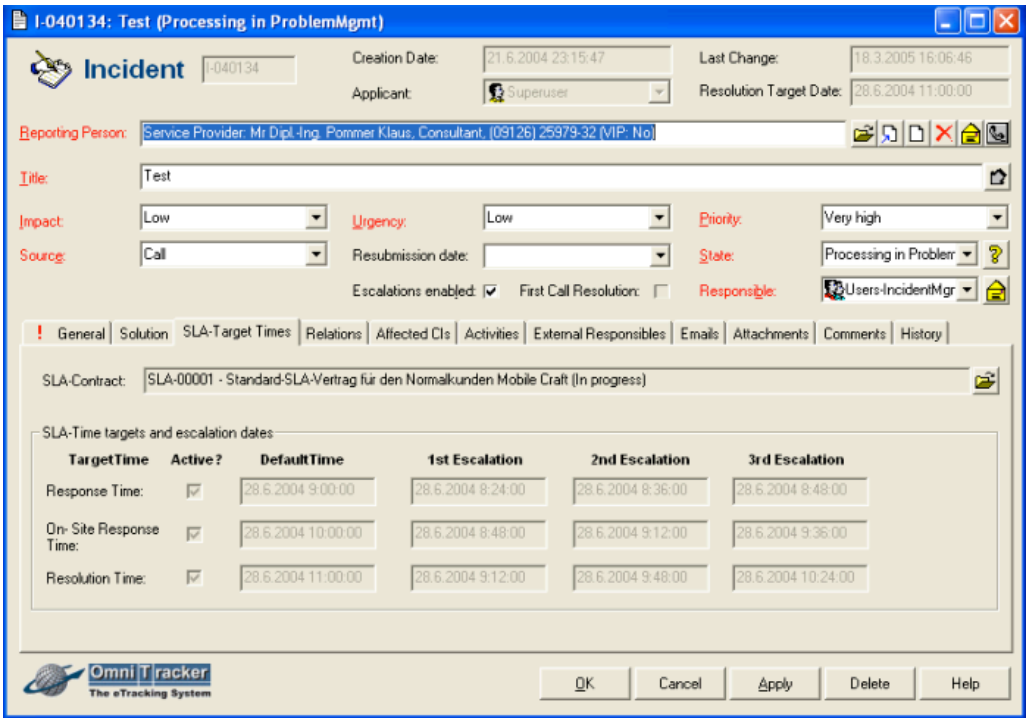
Service Name	Service Users	Configuration Items
StockControl	Mary ...	StockControl Program v1.1 Tomcat v5.5 Oracle 9i Red Hat Enterprise Linux 5 Server Prague Switch1 Switch3 Intranet
Internet	...	Internet Service Provider Firewall Zone F v3.2

Tabulka 4-2: Příklad jednoduchého katalogu IT služeb.

- Adam díky automatickému nástroji pro správu konfigurace zjistí, že disk serveru, na kterém aplikace běží, je plný.
- Zálohuje jinde a smaže na tomto místě dočasné soubory a začne prozkoumávat pouze log pro Oracle DB a Apache Tomcat, jelikož ví z katalogu IT služeb, že služba využívá jen tuto databázi a aplikační server.
- Okamžitě si všimne velikého logu Oracle databáze.
- Zálohuje a vymaže tedy tento log, restartuje pouze danou službu a vyzkouší její funkčnost.
- Okamžitě také připraví skript, který bude pravidelně zálohovat na jiném stroji a poté mazat původní Oracle log (tzv. *workaround*) a instaluje ho. Služba stále funguje dobře.
- Poté vytvoří v Service Desk aplikaci záznam o problému (přiřadí Oracle skupině, která řeší problémy s Oracle databázemi a připojí odkaz na zálohovaný log) – proto, aby se zkoumala a vyřešila kořenová příčina tohoto incidentu, jelikož ji sám neodhalil.
- Nakonec Adam aktualizuje záznam o incidentu a uzavře jej.

[11] Pojem ITIL. Incident je událost, která znemožní pracovat nebo způsobí omezení určité IT služby.

- Mary je o uzavření / vyřešení opět automaticky notifikována e-mailem, takže nyní ví, že může danou IT službu opět využívat.
- Po uzavření incidentu vytvoří Adam záznam ve znalostní bázi (KB) s popisem incidentu, jeho symptomů a přiloží workaround, který daný incident řeší. Tento záznam má pomoci rychle vyřešit incident tohoto typu, pokud se opět někdy v budoucnu objeví.



Obr. 4-3: Příklad záznamu o incidentu v systému OmniTracker

Využívali jsme funkci Service Desk a proces zvaný Incident Management. V této fázi je však vyřešen pouze incident. Viděli jsme, že během několika hodin/možná desítek minut, kdy se okamžitě incidentem začaly zabývat osoby k tomu určené, mohla Mary opět pracovat. Důležitá služba sloužící ke zpracování objednávek tedy není 3 dny nedostupná, zákazníci nemuseli vůbec nic poznat. Nyní je však třeba vyřešit ještě kořenovou příčinu (pojmy ITIL tzv. problém) tohoto incidentu, aby se v budoucnu neopakoval. Podívejme se, jak jej budeme řešit s pomocí procesu Problem Management:

- Je zformován tým Problem Managementu (PrM), jelikož v Service Desku vznikl požadavek na řešení problému.
- Rachel, specialistka na Oracle, je přiřazena k danému problému a začne zkoumat záznam o problému, přidružený incident a také připojený log soubor Oracle databáze.
- Vidí, že v logu se objevují programátorské výpisy.
- Připojí se k Oracle webu, k jejich nástroji na reportování chyb, ale nenajde zde žádnou podobnou chybu.
- Proto přímo zde, v Oracle reportovacím nástroji vytvoří záznam o chybě.
- Po několika dnech je notifikována e-mailem, že Oracle vydal opravnou záplatu, která řeší tento problém.

- Rachel tedy vytvoří požadavek na změnu (RfC – Request for Change), který požaduje a vysvětluje nutnost implementace této záplaty.

Nyní existuje řešení kořenové příčiny a existuje požadavek na implementaci tohoto řešení do provozního prostředí. Za schválení požadavku, otestování a nasazení záplaty jsou zodpovědné procesy Change a Release Managementu. Postup realizace pak vypadal následovně:

- Změna je schválena Change Managerem, jelikož její implementace téměř nic nestojí a řeší kořenovou příčinu v infrastruktuře, tudíž odstraňuje dočasné řešení, tzv. workaround.
- Oracle záplata je otestována v testovacím prostředí, všechny testy proběhnou v pořádku, je tedy možné záplatu nainstalovat i do provozního prostředí.
- Záplata je instalována do provozního prostředí ve večerním okně určeném pro údržbu (od 2.00 do 3.00 v noci) tzv. *maintenance window*.
- Po úspěšné instalaci záplaty je odstraněno dočasné řešení – skript.
- Poté Rachel doplní řešení problému a uzavře záznam.
- Nakonec aktualizuje záznam ve znalostní bázi vytvořený Adamem a přidá k němu odkaz na záplatu řešící tento problém.

Použití formálních, popsaných a automatizovaných procesů definovaných podle ITIL umožnilo provést všechny nutné aktivity mnohem efektivněji než ad hoc přístup v prvním případě.

Na závěr můžeme zdůraznit několik bodů. Incident byl vyřešen mnohem dříve, než v prvním případě. Řešili ho lidé k tomu určení a jeho řešení neovlivnilo práci jiných lidí v IT oddělení (např. programátorů). Tito lidé věděli, co a jak mají dělat. Navíc hned ten samý den proběhlo zkoumání a řešení kořenové příčiny, z důvodu zamezení opakujících se incidentů a z důvodu strukturálního řešení, ne jen dočasného workaroundu.

Automatizované nástroje usnadnili spoustu práce s diagnostikou a hledáním. O všem existují záznamy v nástroji Service Desk, je snadné vysledovat změny a kroky provedené pracovníky podpory. Znalostní báze (KB) může pomoci při příštím řešení podobných incidentů/problémů.

5 Správa IT služeb

V této kapitole se dozvíte:

- Co je to IT služba?
- Jaké metody a postupy existují v této oblasti?

Po jejím prostudování byste měli být schopni:

- Porozumět pojmu IT služba.
- Vyjmenovat procesy ITIL a jejich účel.

Klíčová slova této kapitoly:

IT služba, ITIL, IEEE 1219, Cobit.

Doba potřebná ke studiu: 3 hodiny

Průvodce studiem

Kapitola představuje oblast provozu a údržby vyvinutého software/informačních systémů pomocí standardních postupů, konkrétně pomocí procesů definovaných frameworkem pro provoz a údržbu ITIL, dále standard IEEE 1219 a governance framework Cobit. Nedílnou součástí tématu je také definice pojmu IT služba.
Na studium této části si vyhraďte 3 hodiny.

V dnešní době častých a rychlých změn je nutné, aby organizace upustily od orientace na provoz a poskytování hardwaru, softwaru, infrastruktury a staly se poskytovateli IT služeb podporující podnikání organizací přesně podle jejich potřeb. Poskytování IT služeb je v IT odvětví stále ještě chápáno pouze jako oblast outsourcingu. Interní IT oddělení pracují a fungují stále jako nákladová centra poskytující hardware a jeho správu, stejně tak jako software a jeho správu. Uživatel se pak musí sám snažit dostat k IT týmům a sdělit, co vlastně potřebuje, pokud možno v IT řeči. IT již dávno není chápáno jako konkurenční výhoda, ale jako nutnost, proto je třeba zviditelnit výdaje na IT, jejich smysl a účelnost. Poslední výzkumy rozpočtů ukazují, že výdaje na provoz a údržbu IT neustále stoupají (při téměř stejné hodnotě IT rozpočtů) a tudíž zbývá méně na vývoj nových funkčností IS a IT služeb zvyšující konkurenceschopnost organizace (viz např. [Ga08IT], [Ga09IT]). Ve skutečnosti je třeba, aby nejen poskytovatelé outsourcovaných služeb, ale naopak hlavně interní poskytovatelé IT začaly chápat IT jako celek, jako službu a snažili se ji co nejvíce a flexibilně propojit s potřebami podnikání daného subjektu.

Pojďme nyní představit definici služby podle ITSM instance zvané ITIL®. ITIL® je celosvětově rozšířený framework [Mat] k řízení a poskytování IT služeb a jako takový propaguje propojenost podniku a jeho procesů s IT pokud možno ve formě IT služeb. ITIL® o službě říká následující [ITIL]:

> *„A service is a means of delivering value to clients by facilitating outcomes clients want to achieve without the ownership of specific costs and risks.“*

Česky bychom zmíněnou definici mohli volně přeložit následovně [ITCZ]:

> *„Služba je prostředek dodávání hodnoty zákazníkovi tím, že zprostředkovává výstupy, jichž chce zákazník dosáhnout, aniž by vlastnil specifické náklady a rizika.“*

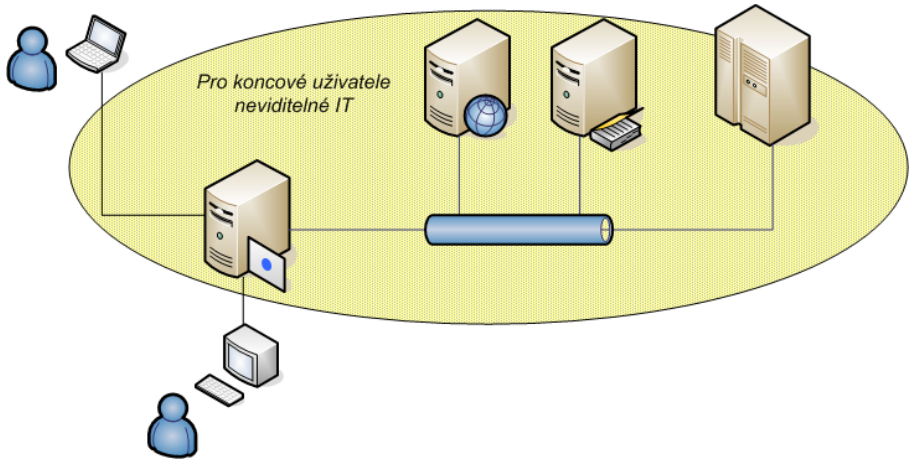
Dva základní principy, které jsou v definici zmíněny, jsou:

- Služba musí podle definice pomáhat uživatelům k dosažení požadovaných (podnikových) cílů.
- Klient si kupuje služby za účelem přenesení rizik a nákladů na odborníka.

Abychom si udělali obrázek, co v praxi taková IT služba je, uvedeme několik příkladů:

- poskytování e-mailu,
- poskytování informací o průběhu doručení poštovní zásilky v kurýrních službách (ať pro potřeby dispečerů a obchodníků, tak pro potřeby vlastního zákazníka),
- podpora monitorování vytíženosti lůžek a operačních sálů v nemocnici (tato služba má přímý dopad na obrat/zisk nemocnice, volné lůžko totiž znamená možnost příjmu dalšího pacienta, IT systém umožní zviditelnit obsazenost a také „poptávku“ na tato lůžka).

Jelikož je typický čtenář tohoto textu technicky orientovaný, je dobré si představit také pozadí IT služby, z čeho všeho se tedy IT služba skládá, co je viditelné pro uživatele, případně pro jakou část IT? Situaci názorně ukazuje následující obrázek:



Obr. 5-1: IT služby a jejich pozadí a také viditelnost pro koncové uživatele

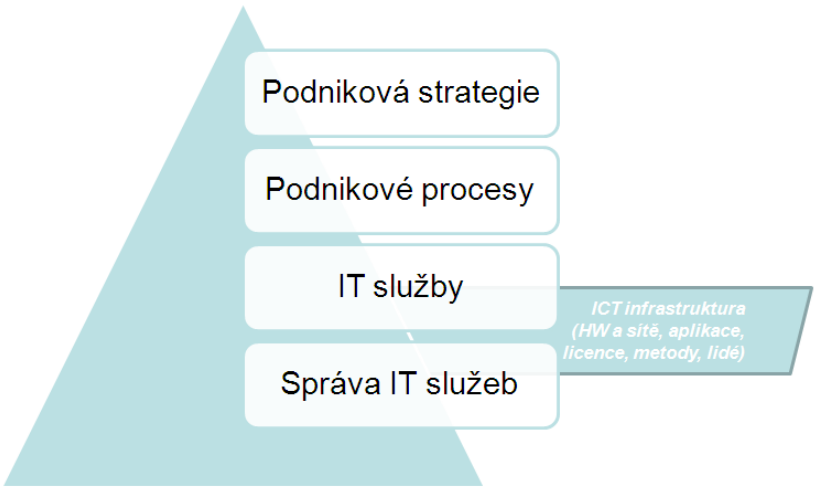
Jak je z obrázku zřejmé, IT služba se skládá z různých komponent, mezi které patří hardware serverů a stanic a software na nich běžící (autorizace a autentifikace, firewally, databáze, middleware, aplikační servery), datová úložiště, mainframe systémy, sítě a síťové prvky nutné k doručení IT služby k zákazníkovi (často také Internet jako páteřní síť) a samozřejmě vlastní aplikace/jejich kombinace či část informačního systému. Všechny tyto

komponenty by měly být běžnému uživateli neviditelné, jakoukoliv chybu či pokles kvality by měl reportovat jako incident[12] spojený s danou službou.

IT naopak technické informace potřebuje. Proto aby tým podpory mohl urychleně vyřešit nějaký incident, který nastal v infrastruktuře, nějaký výpadek či pokles kvality, je třeba vědět, jaké aplikace, síťové prvky či hardware podporují běh dané služby. Jelikož je IT služba konzumována koncovými uživateli s doménovou znalostí a podporována IT profesionály, měla by být popsána z obou pohledů včetně popisu odpovědností jednotlivých stran. IT služby většinou popisujeme v tzv. katalogu služeb (viz příklad v předchozí kapitole).

Cíle a kvalita poskytovaných IT služeb, stejně jako odpovědnosti dodavatele a odběratele služby, způsob měření a reportování výsledků bývá specifikována v dokumentu ***Dohoda o úrovni služeb***, takzvané SLA (*Service Level Agreement*). Tato dohoda bývá většinou součástí kontraktu o poskytování IT služeb, jelikož samotná nemá status smlouvy, ale pouze dohody.

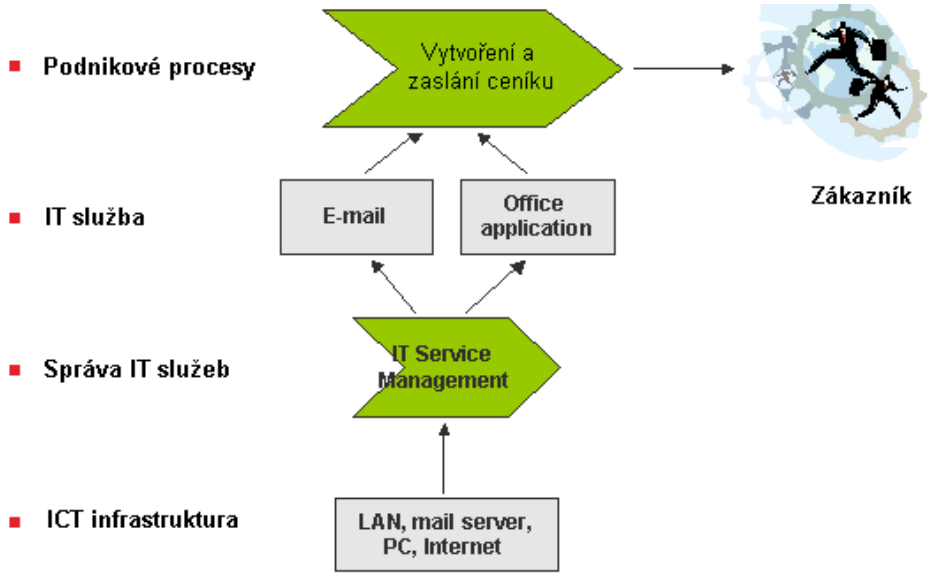
Pokud je tedy cílem IT poskytovat služby, vše z pohledu IT by se mělo točit kolem ***správy a řízení IT služeb*** (anglicky *ITSM – IT Service Management*). Následující obrázek znázorňuje místo IT služeb a jejich správy, podnikových procesů a podnikové strategie. Základem podniku je jeho podniková strategie, která definuje směr, kterým se chce podnik ubírat. Ta také dále říká, co podnik prodává či jaké nabízí služby, na jaké zákazníky se zaměřuje, v jakých lokalitách chce působit, zda bude mít kamenné pobočky či pouze elektronický obchod a spoustu dalšího. Pro realizaci těchto cílů slouží podnikové procesy. Proto, aby mohly být podnikové procesy efektivně vykonávány, používáme IT služby, jež dané podnikové procesy podporují. K provozu těchto služeb využíváme ICT infrastrukturu, aplikace, týmy IT expertů. Samotné služby pak potřebují být také nějakým způsobem spravovány, tj. definovány, nasazovány, provozovány, k tomu slouží správa IT služeb.



Obr. 5-2: Vztah podnikové strategie, procesů, IT služeb a jejich správy.

[12] Incident je událost, která má za výsledek výpadek či sníženou kvalitu poskytované IT služby. Jednou z odpovědností uživatele je reportovat tuto událost dohodnutým způsobem.

Následující obrázek ukazuje toto obecné rozdělení na příkladě. Cílem podnikového procesu nazvaného *„Vytvoření a zaslání ceníku“* je doručit zákazníkovi aktuální ceník produktů. Proto, abychom mohli výstup tohoto procesu uskutečnit, potřebujeme dvě IT služby, konkrétně kancelářskou aplikaci (např. Star Office, StarWriter, StarCalc či MS Word, MS Excel, ...) a e-mail. Tyto IT služby spravuje správa IT služeb a provozovány jsou na ICT infrastruktuře organizace, což je připojení k síti, e-mailový server, PC stanice, na kterých je nainstalována nebo provozována kancelářská aplikace a v neposlední řadě připojení k Internetu, aby mohl být e-mail odeslán. Pro jednoduchost vynecháváme další nezbytné prvky infrastruktury jako jsou switch, e-mailový server apod.



Obr. 5-3: Příklad podnikového procesu a IT služeb.

Ještě uvedeme jeden důvod, proč je vhodné uvažovat v kontextu IT služeb, a ne v kontextu oddělených technických položek jako je hardware, software, sítě apod. Pokud mluvíme o IT službě, máme na mysli veškeré technologie nutné k běhu služby, ale také další nedílné součásti a aspekty, jako jsou licence na softwarové produkty či lidé provozující a podporující aplikaci. Součástí definice služby jsou také vlastnosti a cíle jako náklady na provoz a podporu, doba provozu a podpory, kvalitativní parametry, rozpočet na změny. Proto je také pro IT vhodné mluvit o službách, jelikož jsme schopni vyčíslit konkrétní náklad na danou službu a její provoz, jsme schopni ji spojit s určitými parametry a podnikovým procesem.

V následujících kapitolách se budeme zabývat konkrétními metodami a postupy správy IT služeb. Prvním z nich je de facto standard v této oblasti ITIL.

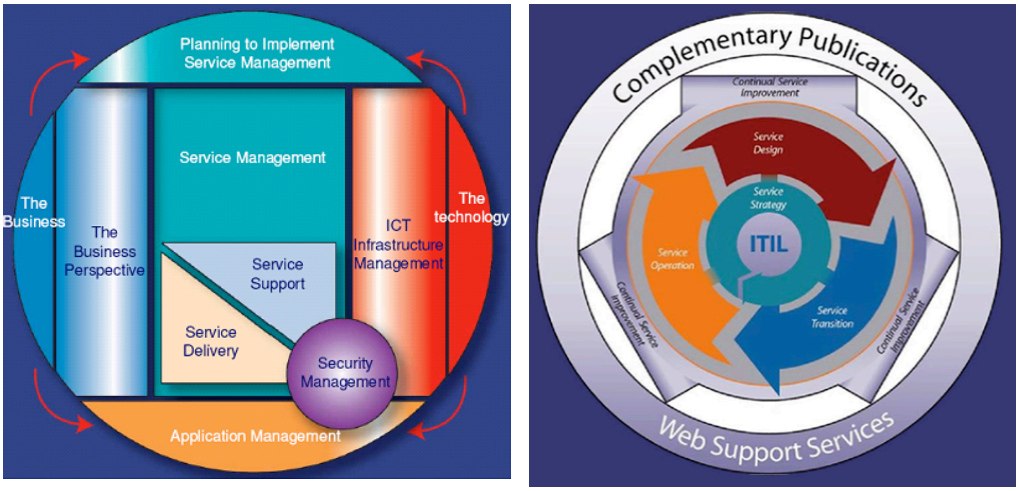
5.1 ITIL®

Zkratka ITIL® znamená *IT Infrastructure Library*, anglicky mluvícím je tedy zřejmé, že se jedná o nějakou knihovnu, která nám dává doporučení co a kdy dělat při provozu a údržbě IT služeb. Hned na začátku zmíníme, že ITIL neříká

jak toto dělat, pouze říká co a kdy. Základním konceptem, kolem kterého je ITIL® postaven, je ***IT služba***. Tento koncept jsme popsali výše.

Druhým pojmem, který je pro ITIL® klíčový, je ***proaktivní přístup***. Klasický reaktivní přístup pouze reaguje na události, které nastaly. Řešení incidentu nebo nutnost změny v infrastruktuře jsou typickým příkladem. Oproti tomu proaktivní přístup se zabývá aktivní detekcí a řešením možných problémů, potřebných změn v ICT infrastruktuře, které by v budoucnu mohly vyvolat incidenty či přinést problémy.

ITIL® je frameworkem sloužícím ke správě IT služeb. Je to pouze framework, není metodikou, říká pouze co a kdy dělat, ale již neříká, jak toto dělat. ITIL® vznikl jako framework postavený na nejlepších zkušenostech z praxe (tzv. *best practices*), na základě úspěšných projektů, stejně jako RUP v případě vývoje software. ITIL® definuje terminologii, jeden jazyk, aby lidé označovali stejné věco stejně a rozuměli si. Dále ITIL® definuje procesy, jejich aktivity a role. Konkrétně je ITIL® knihovnou publikací (knihy, CD) obsahující best practices pro správu IT služeb. Procesy zahrnuté v ITIL byly původně vyvinuty pro britskou vládu, nyní jsou de facto (ITIL®) i de jure (ISO 20000) standardem používaným celosvětově.



Obr. 5-4: ITIL® framework, vlevo v2, vpravo v3 (zdroj: ITIL).

První verze ITIL® z konce 80. let měla 46 knih, přepracovaná verze 2, která spatřila světlo světa kolem roku 2000 má již pouze 8 knih a aktuální opravená verze zvaná ITIL® 2011 (respektive verze 3) má 5 knih podle životního cyklu a 1 přehledovou knihu. Klíčovou změnou ve verzi 3 je nový model životního cyklu. V textu se budeme věnovat popisu verze 2 i verze 3, jelikož pro dokreslení role a smyslu je vhodné použít spíš verzi 2, ale aktuální koncepty jsou ve verzi 3.

Jak již bylo zmíněno, ITIL® není standardem, vůči kterému je možné organizaci posuzovat a certifikovat. Tímto standardem je buď původní britská norma BS 15000 přepracovaná v roce 2003 nebo mezinárodní norma ISO 20000. Obě normy jsou na ITIL založené, vycházejí z něj. Pokud mluvíme o

certifikacích, měli bychom se zmínit o třech možných úrovních certifikování, a to lidí, podniků a nástrojů.

Certifikace lidí (3 úrovně):

- Základní porozumění ITIL (ITIL Foundation Certificate in ITSM) – teorie a pojmy ITIL, základní pochopení.
- Expert – člověk má hluboké porozumění v celé oblasti, může definovat procesy, pracovat jako konzultant.
- Zodpovědné 2 nezávislé certifikační autority EXIN a ISEB.

Certifikace podniků (procesů):

- ISO 20000 – certifikace probíhá pro každý proces zvlášť.
- BS 15000 – původní britská norma.

Certifikace nástrojů (podporujících ITIL procesy):

- Zodpovědná organizace PinkElefant
- Existují verifikační šablony[13] – žádný z nástrojů není verifikován na všech 10 procesů, nejlepší a nejrozsáhlejší (BMC Remedy, CA Unicenter ServiceDesk, HP Service Center a další) vyhovují 6-7 procesům SS a SD.

Struktura ITIL verze 2 se skládá ze sedmi knih:

- Business Perspective – poskytuje IT rady a návody pro lepší porozumění a přispění k cílům byznysu a jak služby lépe přizpůsobit a využít pro maximalizaci přínosů.
- Application Management – popisuje správu aplikací v průběhu celého životního cyklu. Bohužel nevyužívá ověřených praktik v oblasti vývoje softwaru, proto je vhodnější využívat metody k tomu přímo určené jako je RUP, OpenUP, MDIS apod.
- Service Delivery – pokrývá procesy potřebné pro plánování a dodávku kvalitních IT služeb. Zaměřuje se na procesy s delším časovým dopadem, spojené se zlepšováním kvality dodávaných IT služeb.
- Service Support – popisuje procesy spojené s každodenními aktivitami podpory a údržby poskytovaných IT služeb.
- Security Management – popisuje proces plánování a správy definované úrovně bezpečnosti informací a služeb IT. Zahrnuje také analýzu a správu rizik a zranitelností a implementaci nákladově zdůvodněných protiopatření.
- ICT Infrastructure Management – pokrývá všechny aspekty správy ICT infrastruktury od identifikace požadavků byznysu, přes nabídku až k testování, instalaci a následným operacím a optimalizaci komponent IT služeb.
- Planning to Implement Service Management – zaměřuje se na problémy a úkoly související s plánováním, zaváděním a zlepšováním procesů správy IT služeb. Zahrnuje také problematiku kulturních a organizačních změn, rozvoj vize a strategie.

[13] Šablony viz https://www.pinkelephant.com/en-PH/ResourceCenter/PinkVerify/

Jádrem ITIL verze 2 jsou knihy Service Support a Service Delivery. Mezi většinou knih existuje vzájemný přesah.

Struktura ITIL verze 3 se skládá z pěti knih + jedné úvodní:

- Service Strategy – zabývá se sladěním byznysu a IT, strategií správy IT služeb, plánováním.
- Service Design – řešení IT služeb, návrh procesů (tvorba a údržba IT architektury, postupů).
- Service Transition – předání IT služby do byznys prostředí.
- Service Operation – doručení a řídící aktivity procesu, správa aplikací, změn, provozu, metrik.
- Continual Service Improvement – hnací body zlepšení IT služeb, oprávněnost vylepšení, metody, praktiky, metriky.
- Official Introduction of the ITIL Service Lifecycle – základní koncept ITSM, místo ITIL v něm, nový model životního cyklu.

Verze 3 definuje nový životní cyklus IT služby od jejího plánování, přes doručení a zavedení, až po provoz a nestálé zlepšování. Touto problematikou se verze 2 nezabývala, resp. nebyla řešena jako životní cyklus, ale pouze v rámci některých procesů Service Delivery.

Na závěr této kapitoly si ještě shrneme, co ITIL je a co není, co řeší a co neřeší. ITIL® je procesním frameworkem sloužícím pro definici procesů v oblasti správy IT služeb. Vznikl jako soubor ověřených praktik z úspěšných projektů. Nevychází tedy z teoretického prostředí univerzit a konzultantů, ale přímo z praxe. Definuje činnosti a procesy, které je třeba vykonat (a říká kdy) a k nim zodpovědné role. ITIL® však neřeší, jak konkrétně tyto činnosti provádět, jakou mít organizační strukturu. Tyto aspekty ITIL nedefinuje, protože je každá organizace jiná, každá má jinou kulturu a zvyky. Řekli jsme, že ITIL® není metodikou, ale procesním frameworkem, ze kterého si podle našich potřeb metodiku tvoříme. ITIL® není ani standardem, tím jsou britská BS 15000 a mezinárodní ISO 20000. V poslední řadě ITIL® neřeší a neobsahuje metodiku projektového řízení pro nasazení IT služeb, pouze doporučuje metodiku PRINCE2®.

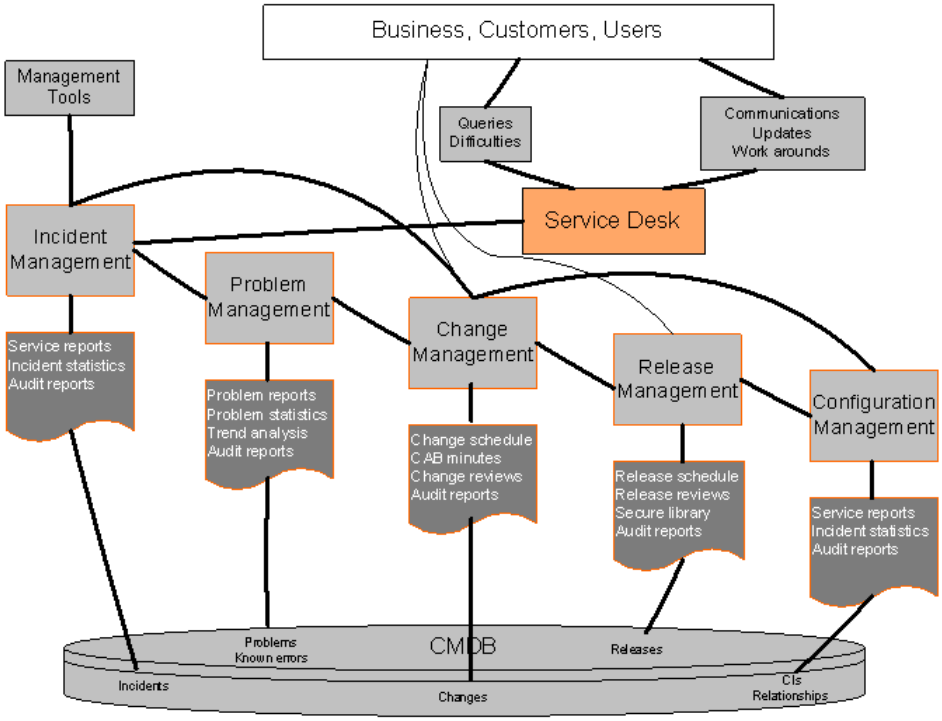
5.2 Stručný přehled procesů ITIL®

V příkladu z kapitoly 4.2 jsme se již s některými ITIL® procesy setkali. Nyní se budeme věnovat popisu procesů vlastního jádra ITIL, tj. pouze dvou základním knihám verze 2: provozu služeb (Service Support) a doručení služeb (Service Delivery). Tyto procesy existují i ve verzi 3 a jsou doplněny dalšími.

Service Support definuje následující procesy:

- Incident Management (IM) – správa incidentů,
- Problem Management (PrM) – správa problémů,
- Change Management (CxM) – správa změn,
- Configuration Management (CoM) – správa konfigurací,
- Release Management (ReM) – správa releasů.
- Funkci Service Desk.

Ústředním bodem Service Support (dále SS) procesů je funkce Service Desku, o které jsme mluvili již v příkladu. ***Service Desk*** je jediným kontaktním místem pro zákazníky a uživatele a má vazbu na všechny, resp. většinu operativních procesů. Service Desk je vlastně aplikace obsahující evidence procesů, tyto evidence mohou být nazájem propojeny a záznamy na sebe odkazovat, např. problémy na incidenty, jak bylo ukázáno v našem příkladu. Cílem Service Desku je zajišťovat každodenní kontakt se zákazníky, uživateli, pracovníky IT a externí podpory. Zajišťuje obnovu výpadku IT služeb a plní roli podpory 1. úrovně a koordinuje 2. a 3. úroveň podpory[14].



Obr. 5-5: ITIL® procesy pro Service Support**.**

Proces ***Incident Management*** je zodpovědný za obnovení normálního provozu IT služby při problémech či výpadku. Snahou je nejen co nejrychlejší obnova služby, ale také minimalizace důsledků výpadku na provoz, resp. na byznys – zákazníka a uživatele. Cílem IM je také zajistit, aby byly služby dodány zákazníkům v kvalitě, která byla dohodnutá v SLA dokumentu zmíněném výše. Odpovídá tedy za správu všech incidentů od zjištění a vytvoření záznamu, až po vyřešení a uzavření.

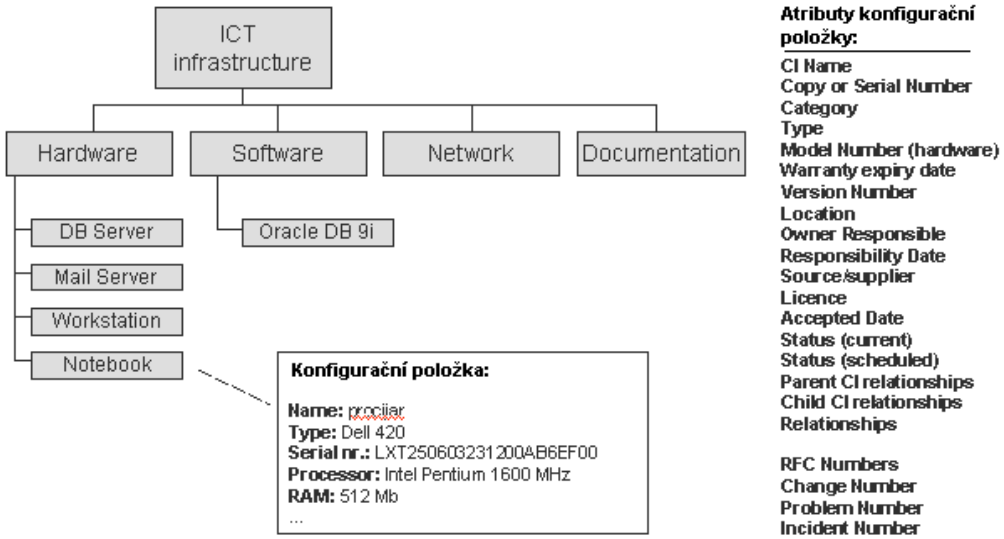
Cílem procesu ***Problem Management*** je minimalizovat nepříznivé dopady incidentů a problémů na byznys. PrM asistuje IM při řešení závažných incidentů. PrM je zodpovědný za evidenci náhradních řešení (tzv. *workaround*), rychlých náprav jako známých chyb, a tam kde je to vhodné

[14] 1. úroveň podpory – aplikační specialisté, řeší také pomoc v případě dokumentace, instalace.
2. úroveň podpory – techničtí specialisté, řeší technické problémy konkrétních aplikací, které nedokážou specialisté první úrovně vyřešit.
3. úroveň podpory – specialisté na daný produkt, většinou třetí strany (výrobci), ti už odstraňují chyby přímo v kódu nebo zařízeních a řeší problémy, které ani jedna z předchozích vrstev nevyřešila.

(finančně efektivní), iniciuje změny trvale implementované do infrastruktury – strukturální změny. PrM rovněž analyzuje incidenty a problémy a zkoumá jejich trendy, aby proaktivně zabránil jejich dalšímu výskytu.

Pro efektivní zpracování a implementaci změn slouží proces ***Change Management***. Změny jsou pomocí procesu řízeny od zaznamenání záznamu, přes filtraci, posouzení smyslu, kategorizaci až po plánování, testování a nasazení. Jedním z klíčových výstupů procesu je plán změn (anglicky *Forward Schedule of Change*), který obsahuje program změn dohodnutý se všemi odděleními, zákazníky a vytvořený podle dopadu na byznys.

Proces ***Release Management*** řeší změny IT služeb z pohledu globálních souvislostí. Zabývá se aspekty nasazení, akceptace a distribuce software a hardware. ReM je zodpovědný za sestavení, testování a distribuci releasů a také za jejich bezpečné uložení, k tomu slouží úložiště originálních verzí softwaru zvané *Definitive Software Library* (DSL) a *Definitive Hardware Store* (DHS) pro hardwarové komponenty a zařízení.



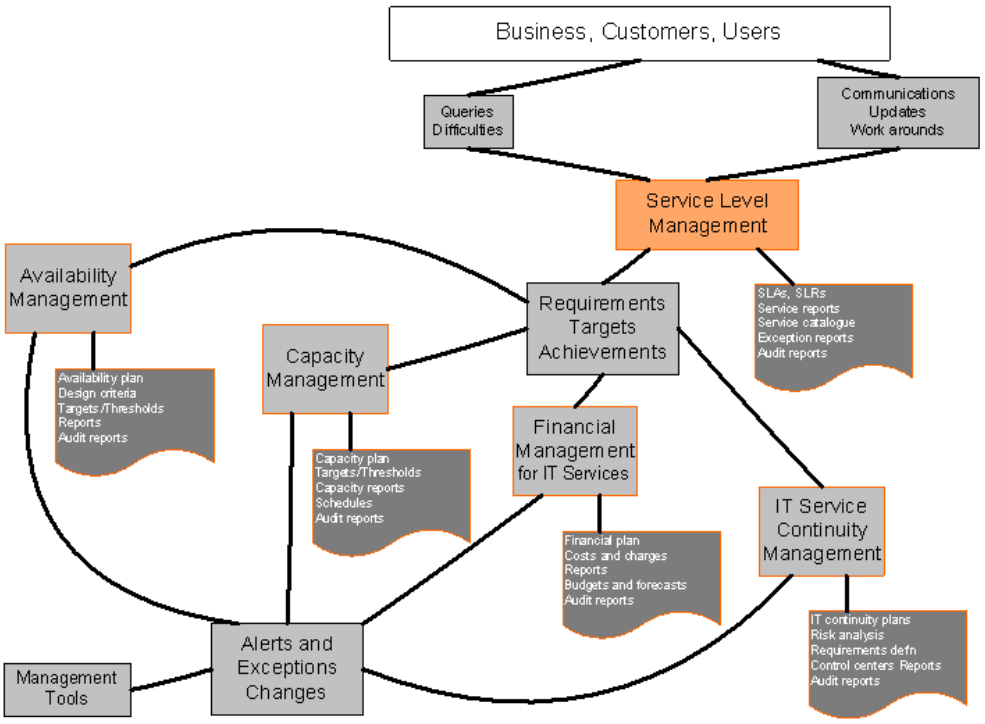
Obr. 5-6: Struktura konfiguračních položek, příklad atributů.

Základnou, na které jsou postaveny ostatní procesy ITIL®, je proces ***Configuration Management***. Proces sám o sobě není příliš viditelný, ale poskytuje nezbytné informace ostatním procesům. Základním prvkem tohoto procesu je tzv. konfigurační databáze (CMDB – *Configuration Management Database*), která se skládá z jedné nebo více fyzických databází či pohledů. CMDB obsahuje detailní záznamy o jednotlivých komponentách ICT infrastruktury. Tyto prvky jsou označované jako konfigurační položky (CI – *Configuration Items*). Hlavním přínosem této databáze je zachycení vzájemných vztahů konfiguračních položek, což umožňuje nejen modelovat scénáře IF-THEN (co se stane, když ...), ale také v případě výpadku konkrétního prvku dohledat možné komplikace pro další prvky.

Service Delivery definuje následující procesy:

- Service Level Management (SLM) – správa úrovně služeb,
- Capacity Management (CaM) – správa kapacit,

- Availability Management (AvM) – správa dostupnosti,
- IT Service Continuity Management (ITSCM) – obnova IT služby po výpadku,
- Financial Management for IT Services (FiM) – správa financí IT služeb.



Obr. 5-7: ITIL® procesy pro Service Delivery (zdroj: ITIL).

Procesem zodpovědným za dojednání kvality IT služeb je ***Service Level Management***. SLM sbírá a hodnotí požadavky byznysu na IT služby. Na základě těchto požadavků vytvoří a provozuje IT službu a údaje o její kvalitě sepisuje v dohodě o úrovni služby (SLA – *Service Level Agreement*). Pro podporu vlastního SLA je možné ještě dohodnout podpůrné smlouvy OLA – *Operational Level Agreement* mezi interními odděleními a nebo externí UC – *Underpinning Contract*. Pouze druhý jmenovaný má status právního dokumentu, ostatní jsou pouze dohody, musí (měly by být) být tedy přílohou nějaké smlouvy o poskytování IT služeb. SLM proces je také zodpovědný za definici katalogu a údržbu služeb, který popisuje jednotlivé IT služby. Katalog popisuje zákazníky, kteří si mohou služby objednat a využívat je. Pro potřeby IT pak katalog definuje, které prvky ICT infrastruktury jsou využívány pro kterou IT službu. Díky tomuto seznamu, který je publikován prostřednictvím Service Desku a dostupný všem jeho uživatelům, je pak při výpadku některé konfigurační položky jasně vidět, které služby jsou ohroženy. Uživatelé o tom mohou být okamžitě obeznámeni a může být sjednána náprava. Zde je jasná vazba na Service Desk a jeho funkce. Následující obrázek ukazuje příklad katalogu služeb:

Service Name	Description	Service Users		Configuration Items
StockControl	Service provides functions for order processing,	Trade dept. Warehousing dept. Marketing dept.		StockControl Program v1.1 Tomcat v5.5 Oracle 9i Red Hat Enterprise Linux 5 Server Prague Switch1 Switch3 Intranet
Internet	Service provides connection to Internet	All users		Internet Service Provider Firewall Zone F v3.2
Document Search	Service allows searching for documents	All users		Google Desktop Search

Obr. 5-8: Příklad katalogu IT služeb. Sloupec úplně vpravo obsahuje seznam konfiguračních položek viditelných jen pro IT.

Proces ***Financial Management for IT Services*** umožňuje provozovat IT jako byznys. Základní aktivity zahrnují porozumění nákladům a jejich účtování, prognózy budoucích finančních plánů a další. Proces definuje jednu volitelnou položku: účtování za IT služby, jež se snaží o navrácení nákladů na provoz IT z byznysu spravedlivým a poctivým způsobem.

Proces ***Capacity Management*** se zabývá dostatečnou, resp. akorát potřebnou kapacitou k uspokojení požadavků a cílů byznysu. Pro tento účel vytváříme a postupujeme podle kapacitního plánu (CP – *Capacity Plan*), který je úzce spjatý s plány a cíly byznysu. CP pokrývá tři oblasti – správa kapacit z pohledu byznysu, IT služeb a zdrojů. Pro dosažení cílů používá proces aktivity, jako je řízení výkonnosti služeb (*Performance Management*), správa požadavků uživatelů podle času a zatížení (*Demand Management*) nebo škálování a modelování aplikací (*Application Sizing and Modelling*). Z poslední aktivity je zřejmé, že tento tým by měl spolupracovat s vývojovým týmem minimálně v otázkách rozšiřitelné architektury a výkonnosti vyvíjené aplikace.

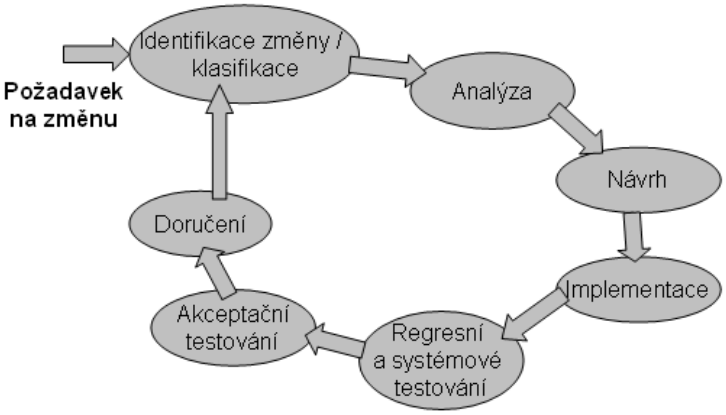
Proces ***IT Service Continuity Management*** produkuje plány obnovy, které jsou navrženy tak, aby po každém velkém výpadku / incidentu byly služby poskytovány na dohodnuté úrovni v dohodnutém čase (jak je dohodnuto v SLA dokumentu). Tento proces je součástí podnikového plánu zachování chodu byznysu (BCP – *Business Continuity Plan*) při nějakém významném problému, jako jsou například povodně či požár. Cílem procesu ITSCM je napomoci byznysu minimalizovat narušení základních podnikových procesů během závažného incidentu a po něm. V rámci procesu jsou také pravidelně prováděny aktivity jako analýza dopadu na byznys, analýza rizik, testování a údržba plánů obnovy.

Klíčovým aspektem IT služeb je logicky jejich dostupnost. Proces ***Availability Management*** je odpovědný za splnění požadované dostupnosti IT služeb či za jejich překročení (vyšší hodnoty), stejně jako jejich proaktivní zlepšování. Pro

dosažení těchto cílů proces monitoruje, měří, reportuje a vyhodnocuje klíčové metriky každé služby a jejich součástí. Mezi ně řadíme dostupnost (*availability*), spolehlivost (*reliability*), udržovatelnost (*maintainability*), praktičnost (*serviceability*) a bezpečnost (*security*).

5.3 Standard IEEE 1219

Standard mezinárodní organizace IEEE 1219 nazvaný *Standard for Software Maintenance* organizuje proces údržby do sedmi fází (viz obrázek). Kromě vlastních fází a jejich pořadí definuje standard také vstupy a výstupy fází, aktivity, podpůrné procesy a sady metrik. Standard se však zaměřuje pouze na problematiku údržby (tvorba nových funkčností, úprava a oprava existujících), samotný provoz (denní monitoring, zálohy, řešení incidentů apod.) neřeší.



Obr. 5-9: Proces údržby podle IEEE 1219.

Následující text stručně shrnuje jednotlivé fáze standardu [IEEE2]:

- Identifikace problému, změny, klasifikace a prioritizace (*Problem, modification identification, classification*) – uživatel, zákazník, vývojář či manažer vytvoří požadavek na změnu, který je přiřazen do dané kategorie (korekce, přizpůsobení, zdokonalení – viz kap 4), kde je mu přidělena priorita a unikátní identifikátor. Fáze obsahuje také aktivity pro ohodnocení požadavku (zda ho zamítnout či přijmout) a jeho přiřazení do plánu implementace.
- Analýza (*Analysis*) – náplní této fáze je identifikace počátečního plánu návrhu, implementace, testování a doručení. Analýza je prováděna ve dvou rovinách:
  - proveditelnost – nalezení alternativních řešení a posouzení jejich dopadu a ceny,
  - detailní analýza – identifikace a detailní specifikace požadavku na změnu, návrh testovací strategie, sestavení implementačního plánu.
- Návrh (*Design*) – fáze slouží pro návrh konkrétních změn existující aplikace s využitím existující dokumentace, software, databází a výstupů z analýzy. Aktivity zahrnují zahrnují následující:
  - identifikace dotčených softwarových modulů/komponent,
  - modifikace relevantní dokumentace těchto modulů/komponent,
  - vytvoření test casů nového návrhu,

    - identifikace regresních testů.
- Implementace (*Implementation*) – fáze zahrnuje kódování, unitové testování, integraci modifikovaného kódu, integrační a regresní testování, analýzu rizik a revize.
- Regresní/systémové testování (*Regression/system testing*) – záměrem fáze je testování celého systému s cílem ověřit naplnění původních a modifikovaných požadavků a zachycení případných zavlečených chyb v době implementace změny. Jedním z cílů fáze je také připravit systém pro následující fázi akceptačního testování.
- Akceptační testování (*Acceptance testing*) – tato fáze se zaměřuje na zákaznické testování (možné provádění třetí stranou) změny plně integrované v systému a zahrnuje funkční testy, regresní testy a také testy integrační.
- Doručení (*Delivery*) – cílem fáze je doručení a instalace modifikovaného systému/změny do provozního prostředí. Tato fáze zahrnuje aktivity:
    - informování zákazníka a uživatelů,
    - instalace,
    - školení,
    - příprava záložní verze.

Ze stručného přehledu obsahu a struktury standardu je zřejmá jeho procesní orientace – fáze, aktivity, dokumenty, vstupy, výstupy, role. Standard se zaměřuje pouze na problematiku údržby, neřeší problematiku provozu. Přínosem standardu je určitě zaměření se na možné varianty řešení a hodnocení jejich dopadu a ceny. Již však neříká, jak tohoto využít v komunikaci se zákazníkem a jak a zda diskutovat změnu požadavku, abychom využili stávající funkčnosti v co největší míře. Druhým pozitivním bodem je testování, kdy například již při návrhu je zdůrazněno hledání vhodné testovací strategie. Detailní popis standardu IEEE 1219 viz [IEEE2].

5.4 CobiT®

Pro úplnost představíme také nástroj pro strategické řízení IT. CobiT® je zkratkou z anglického *Control Objectives for Information and related Technology*. Jedná se o framework pro IT governance, tj. strategické směrování IT s cílem správně podpořit byznys a tudíž také nastavení podmínek (a metrik) pro toto směřování v kontextu organizace. Čtenáře upozorníme, že CobiT® nám může pomoci s mapováním IT na strategii firmy. Jak má vlastně IT podporovat dosahování strategie. O této problematice jsme se podrobně bavili v předmětu softwarové inženýrství a CobiT® je jedním z nástrojů, který toto propojení efektivně umožňuje. CobiT slouží převážně lidem z byznys oblasti. Na rozdíl od ITIL či MANTEMA se nezabývá denodenními činnostmi IT. Jedná se o nástroj pro management organizace umožňující propojit cíle organizace s jednotlivými IT projekty a to dokonce až na úroveň praktik (přístup, který je dnes ražen například IBM a jejich MCIF[15]). V roce 2012 vyšla verze CobiT® 5.

[15] IBM MCIF® – IBM Measured Capability Improvement Framework je řešení umožňující mapovat podnikové cíle na jednotlivé praktiky vývoje software a také je pravidelně měřit. Pro

Struktura CobiT je následující. Je definováno 34 procesů, které jsou rozděleny do 4 oblastí, domén:

- Plánování a řízení (PO – *Plan and Organise*),
- Pořízení a implementace (AI – *Acquire and Implement*),
- Doručení a podpora (DS – *Deliver and Support*),
- Monitorování a hodnocení (ME – *Monitor and Evaluate*).

Každý proces dané oblasti je popsán pomocí svého účelu a cíle, kontrolních cílů, definice vstupů a výstupů, klíčových aktivit procesu a rolí a také pomocí metrik. CobiT definuje podnikové cíle, které mají přímý vztah k IT a seskupuje je do 4 základních kategorií, které jsou mimo jiné známé z Balanced Scorecard. ***Dále mapuje tyto podnikové cíle na strategické IT cíle***. Tímto přístupem jsme schopni zajistit propojení denních aktivit v oblasti IT s cíli organizace a také je díky existujícím metrikám měřit a reportovat vedení.

Příkladem těchto cílů a jejich mapování je následující ukázka:

	Byznys cíle	**IT cíle**			
1	Zvýšení podílu na trhu	10	16		
2	Růst příjmů	10	16		
3	Řízení provozních rizik	2	8	9	10
4	…	…			

	IT cíle	**IT Procesy**			
1	…	…			
	…	…			
10	Realizace projektů v plánovaném čase, kvalitě a rozpočtu	PO8	PO10		
…		…			
16	Zajištění efektivních služeb a připravenost na změny	PO5	DS6	ME1	ME3
...	…	…			

Tabulka 5-1: Mapování byznys cílů na IT cíle a IT procesy podle CobiT®.

Jak je zřejmé, je CobiT zaměřen na trochu jinou oblast než na popis aktivit pro potřeby provozu, údržby a podpory. Primárně slouží manažerům a auditorům. Těm prvním k tomu, aby mohli propojit IT projekty s podnikovými cíli a také sledovat jejich plnění. Druhým pak pomáhá hodnotit využití finančních prostředků investovaných do IT v rámci organizace. Pro úplnost jsme jej však uvedli, jelikož poskytuje nástroje k propojení IT a podnikových cílů, což je stále jednou z největších slabin poskytování IT služeb.

5.5 Shrnutí přístupů k provozu a údržbě IT

Jak je viditelné, u standardů (např. ISO a IEEE procesy) a doporučení se jedná o definice procesů, aktivit, procedur, které by měly provádět určité role. Lidský aspekt je pominut respektive není výrazným předmětem těchto definic.

---

tento účel je poskytováno několik nástrojů, které výrazně ulehčují sběr dat a vyhodnocení metrik. Více viz [Kr07] a [Kr08].

Závěrem tedy můžeme shrnout situaci v oblasti standardů, metodik a přístupů k provozu a údržbě IT následovně:

- Existují dobře zavedené, využívané a rozhodně ne špatné přístupy.
- Přístupy jsou i u nás docela rozšířené (např. [Mat] deklaruje 63% společností v ČR využívajících ITIL, srovnej například se Skandinávií, kde je použití ve více než 70%).
- Ve všech přístupech je viditelná snaha o propojení byznysu s IT, vývoje s údržbou, snaha o vysvětlení vzájemných potřeb, sjednocení jazyků.
- Přístupy jsou však příliš procesně orientované, opomíjejí lidský aspekt, předpokládají, že díky procesu budou mít stejný výsledek rozdílné týmy (stejně jako tomu bylo v případě vodopádového přístupu ve vývoji software v 80. letech minulého století).

Existence mnoho různých metod, standardů či frameworků znesnadňující jejich čitelnost, využitelnost a relevantnost.

Kontrolní otázky:

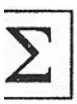
1. Co je to ITSM (správa IT služeb)?
2. Vyjmenujte alespoň dva procesy Service Supportu.
3. Vyjmenujte alespoň dva procesy Service Delivery.
4. Jaký proces se zabývá co nejrychlejším vyřešením incidentu?
5. Jaký je rozdíl mezi incidentem a problémem?
6. Co je to IT služba?

Úkoly k zamyšlení:

Zamyslete se nad možnostmi zálohování existujících softwarových služeb, jaké všechny Vás napadnou (zálohy dat, záložní servery, prostory pro okamžitou instalaci, pojištění, záložní centra, ...). Jaký si myslíte, že bude rozdíl mezi vybranými a použitými možnostmi v bance, průmyslovém podniku a střední IT firmě?

Korespondenční úkol:

Zmínili jsme stručně problematiku Service Desku, což není jen nástroj, ale je to jediné kontaktní místo pro zákazníky (tzv. SPOC – *Single Point of Contact*). Jelikož toto místo slouží k zaznamenání incidentů, problémů, požadavků na změny, k nahlášení a dohodnutí implementace záplat, aktualizací s uživatelem, je zřejmé, že spokojenost uživatelů bude hodně záviset na lidech, zde pracujících. Představte si že píšete inzerát, pokuste se vypsat, jaké schopnosti a dovednosti (včetně tzv. měkkých dovedností) by měl mít takový pracovník a také jaké bude mít odpovědnosti.



Shrnutí obsahu kapitoly

V této kapitole jste se seznámili s oblastí provozu a údržby doručeného software pomocí standardních postupů (ITSM – *IT Service Management*), konkrétně pomocí procesů definovaných frameworkem pro provoz a údržbu ITIL. Text vysvětlil, co jsou to IT služby, co je to ITIL a na příkladu ukázal jeho aplikaci. V kapitole jsme se také stručně věnovali popisu jednotlivých procesů ITILu a dalších standardů a frameworků z dané oblasti jako je IEEE 1219 a Cobit.

6 Microsoft .NET Framework

V této kapitole se dozvíte:

- Co znamená zkratka .NET
- Jaká je architektura .NET
- Co je třeba pro běh aplikací v .NET
- Co je třeba pro vývoj aplikací v .NET
- Jak se programuje za použití jazyka C#
- Jak se tvoří webové aplikace .NET
- Jaké další technologie jsou spojeny s .NET

Po jejím prostudování byste měli být schopni:

- Nastavit počítač pro spouštění .NET aplikací
- Rozumět principům tvorby .NET aplikací
- Chápat výhody a vlastnosti a vhodně rozhodnout o použití .NET pro vývoj aplikací.

Klíčová slova této kapitoly:

.NET, MS.NET, C#, VB.NET, Common Language Runtime, Silverlight

Doba potřebná ke studiu: 5 hodin

Průvodce studiem

Kapitola představí pojem .NET, architekturu .NET frameworku, důvody jejího vzniku. Ukáže vývojové nástroje a jazyky pro tvorbu aplikací nad .NET frameworkem a požadavky na jejich následné spouštění na počítačích. Dále bude naznačen způsob tvorby aplikací pro .NET framework, jak desktopové tak pro webové řešení a ukázán z působ začlenění nových technologií do .NET
Na studium této části si vyhraďte 5 hodin.

6.1 Představení .NET Frameworku

6.1.1 Co je .NET Framework

Obecně pojem .NET odkazuje na produkt Microsoftu, který zahrnuje prostředí, kompilátory a sadu knihoven pro použití při vývoji aplikací.
Microsoft .NET Framework je softwarová komponenta, která je volitelnou částí operačních systémů Microsoft Windows. Poskytuje velké množství funkcionality často používanými při programování a spravuje spouštění programů, které jsou napsány pro tento framework.
Předkódovaná funkcionalita vytváří ve .NET Frameworku tzv. Base Class Library, tedy knihovnu základních funkčností, a poskytuje řešení pro široké množství programovacích požadavků: uživatelská rozhranní, přístupy k datům, přístup k databázím, kryptografii, vývoj webových aplikací, numerické algoritmy, síťová komunikace a další. Knihovnu BCL používají všichni programátoři při tvorbě aplikace. Samozřejmě, stejně jako u ostatních

podobných technologií, i v .NET programátor může vytvářet a znovupoužívat knihovny vlastní.
Programy napsané pro .NET Framework se spouští v softwarovém prostředí, které spravuje požadavky na běh programu. Toto běhové prostředí (které je tedy také částí .NET Frameworku) se nazývá Common Language Runtime (CLR) a lze si jej představit jako aplikační virtuální stroj. Programátor se tedy nemusí starat o specifika hardwaru nad kterými bude aplikace spuštěna (obecně specifikace, kterou CLR splňuje, je platformě nezávislá), navíc CLR nabízí další důležité služby, jako jsou zabezpečovací mechanismy, správa paměti, správa výjimek, správa knihoven a připojených zdrojů a další. Base Class Library a Common Language Runtime dohromady tvoří základ MS .NET Frameworku.

6.1.2 Tvorba aplikace

Celý MS .NET Framework je postaven na robustní architektuře s ohledem na maximální variabilitu při dalším vývoji nebo změnách funkčnosti frameworku. Snahou byla i možnost lehké rozšiřitelnosti .NET Frameworku s ohledem na existující jazyky. Sama společnost Microsoft vytvořila k .NET Frameworku čtyři základní programovací jazyky (VB.NET, C++/CLI, C# a JScript), ale otevřenost standardu způsobila vznik velkého množství dalších jazyků (od třetích firem), které jsou zpravidla deriváty již existujících (PHP.NET, J#, Python.NET, a další). Jak to tedy funguje?
Pří vývoji aplikace na počátku programátoři vyberou jeden (nebo více jazyků, je možná jejich interoperabilita), ve kterých bude aplikace vyvíjena. Napsaný zdrojový kód ve zvoleném programovacím jazyce se následně kompilátorem daného jazyka překládá do speciálního mezi-jazyka nazvaného MSIL (Microsoft Intermediate Language), ve kterém je nyní aplikace uložena a připravena ke spuštění. Vznikne tzv. assembly – sestavení[16]. Při spuštění aplikace se již inicializuje prostředí .NET Frameworku, které daný MSIL kód přeloží pomocí dalšího kompilátoru (JIT[17]) do strojového kódu daného procesoru a následně tento kód vykoná.

6.1.3 Nasazení aplikace

Aplikace psaná pro .NET Framework ke svému spuštění na cílovém počítači potřebuje (kromě svých knihoven) nainstalovaný běhové prostředí Microsoft .NET Framework, tedy vlastně balíček CLR a přidružených knihoven a souborů, který umožňuje spouštění .NET aplikací. Žádná další konfigurace obecně při nasazení není potřebná (pokud ji nevyžaduje sama nasazovaná aplikace). Nasazovaná aplikace si nese všechny potřebné informace uloženy sebou, a po přeložení ji tedy lze typicky okamžitě spustit na libovolném počítači s nainstalovaným .NET Framework odpovídající verze. Tyto balíčky jsou pro postupně zahrnovány do instalací operačního systému Windows.

[16] Assembly si pro zjednodušení můžete představit jako klasickou knihovnu, kterou lze znovu použít.
[17] JIT – just in time kompilátor

6.1.4 Verze .NET

Platforma .NET prošla během svého relativně krátkého období již několika verzemi. Následuje jejich přehled:

- **.NET Framework 1.0 –** vydán v únoru 2002. První verze .NET Frameworku. Pro vývoj používá Microsoft Visual Studio 2002 (codename Rainier);
- **.NET Framework 1.1** – vydán v dubnu 2003. Jedná se o první velký update předchozí verze. Pro jeho vývoj bylo uvolněno Microsoft Visual Studio 2003 (Everett);
- **.NET Framework 2.0 –** vydána na počátku roku 2006. Obsahuje radikální vylepšení jazyka i knihoven oproti předchozím verzím. Tato verze byla jako první masově rozšířená. Využívá pro vývoj Microsoft Visual Studio 2005 (Whidbey);
- **.NET Framework 3.0** – vydán v listopadu 2006. Obsahuje velké změny co se týče schopností a vlastností programovacích jazyků (zejména C#). Není běžně používán, sloužil jako základ pro framework 3.5. Pro vývoj využívá stále Microsoft Visual Studio 2005;
- **.NET Framework 3.5** – vydán v listopadu 2007. Opět se jedná pouze o rozšíření předchozí verze (i když zásadní). Pro vývoj je uvolněno nové Microsoft Visual Studio 2008 (Orcas), které jako první podporuje vývoj pro více verzí .NET Frameworku.
- **.NET Framework 4.0** – vydán v dubnu 2010. Jedná se o opět zcela nový pohled na programovací jazyky, kdy se doplňuje spousta vlastností týkajících odlišných přístupů, zejména funkcionálního přístupu. Pro vývoj slouží Visual Studio 2010.
- **.NET Framework 4.5** – očekává se vydání v průběhu roku 2012. Hlavním cílem je podpora vývoje aplikací stylu Metro a obecně aplikací určených pro operační systém Windows 8. S příchodem této verze se očekávají velké změny ve vývojovém prostředí.

V současné době se nejčastěji používají verze 2.0 (z historických důvodů), 3.5 (protože pro ni bylo napsáno největší množství aplikací) a 4.0 (pro v současnosti vznikající nové aplikací).

Velkou otázkou je zpětná kompatibilita jednotlivých frameworků. Obecně platí, že frameworky 1.1 a 2.0 nejsou zpětně kompatibilní s předchozími frameworky (ani zpětně). Framework 3.5 a vyšší již zpětně kompatibilní s předchozími verzemi (do verze 2.0) je.

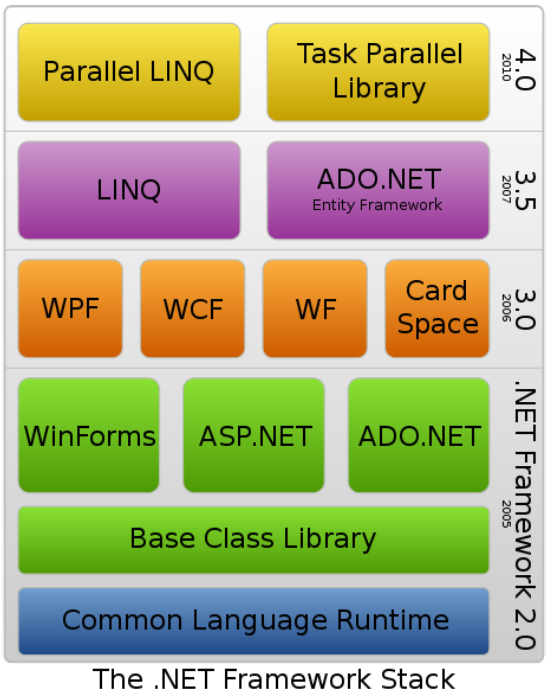
Dalším bodem je dostupnost frameworků – frameworky 1.0, 1.1 a 2.0 nejsou součástí žádné instalace OS v základní verzi, ale lze je doinstalovat. Automaticky se instaluje framework 2.0 pro Windows XP SP2. Framework 3.0 je automaticky instalován v OS Windows Vista a Windows Server 2003. Framework 3.5 (SP1) je automaticky instalován ve Windows 7. Framework 4.0 není v současné době instalován v žádném OS, ale lze jej doinstalovat od verze XP výše.

6.1.5 Edice .NET

.NET framework vychází v několika edicích odlišujících se schopnostmi a obsaženou funkcionalitou s ohledem na cílové zařízení. Běžně se odlišují tyto edice:

- .NET Framework – běžný balík určený k instalaci na klasické počítače a servery od Windows 98 a vyšší;
- .NET Framework Client Profile – odlehčený balík určený k instalaci na klasické počítače pro jednodušší programy (od FW 3.5);
- .NET Compact Framework – pro kapesní počítače a mobilní telefony se systémem Windows Mobile a Windows Phone;
- .NET Micro Framework – pro embedded zařízení s minimálními požadavky na HW.

6.2 Architektura

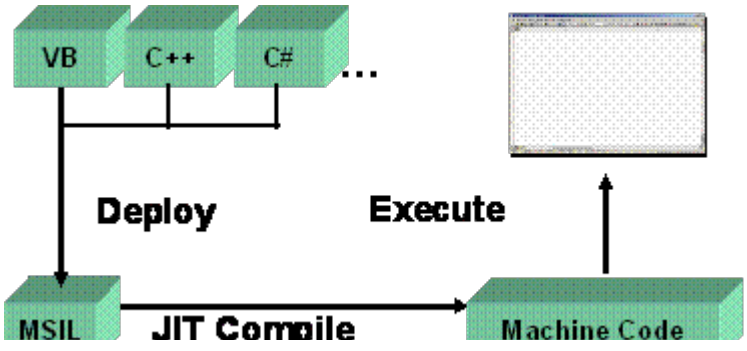


Obr. 6- 1: Architektura .NET frameworku (zdroj: http://en.wikipedia.org/wiki/File:DotNet.svg)

6.2.1 CLI, CLR

Základní aspekty .NET Frameworku leží na *Common Language Infrastructure* (CLI). Jedná se o **specifikaci**, jejímž cílem je poskytnout jazykově nezávislou platformu pro vývoj a spouštění aplikací, včetně funkcionalit pro zachytávání výjimek, grabage collectoru, bezpečnosti a interoperability aj. V původní idee je tedy architektura .NET **multiplatformní**.

Microsoftem vytvořená **implementace** CLI pro .NET Framework je nazývána *Common Language Runtime* (CLR). Jedná se



komponentu virtuálního stroje, která definuje spouštěcí prostředí pro programový kód .NET aplikací. Just-in-time kompilátor (který je součástí CLR) zpracuje předkompilovaný kód MSIL do strojového kódu pro daný procesor a CLR se stará o aplikaci v době jejího spuštění. Tímto postupem je vývojář oproštěn od řešení mnoha problémů při implementaci aplikace a CLR také nabízí některé služby, které ulehčují tvorbu aplikace, zejména správu paměti, správu vláken, správu vyjímek, garbage collector nebo zabezpečení.
Pro alternativní platformy existuje projekt Mono, který zahrnuje platformy derivátů Linuxu, Unixu, Solaris či OS X. Není však přímo podporován firmou Microsoft (v současné době jej zaštituje Novell). Protože však vzniká zpětně po vydání jednotlivých verzí FW, není vždy s danou verzí FW zcela kompatibilní. Proto platforma .NET není uvažována jako plně multiplatformní technologie.

6.2.2 MSIL, Assembly

Výsledkem překompilování aplikace je převedení jejích zdrojových kódu ve zvoleném jazyce .NET (C#, VB.NET nebo jiného) do tzv. mezikódu. Obecně je takovýto mezikód nazýván *Common Intermediate Language* (CIL, nebo pouze IL). Java používá podobné řešení, její mezikód je nazýván bytecode. CIL pro .NET Framework je nazýván *Microsoft Intermediate Language* (MSIL). Tento mezikód je uložen do jednoho nebo více souborů, kterým se říká sestavení – Assembly.
Assembly se může sestávat z jednoho nebo více souborů, ale alespoň jeden soubor musí obsahovat *Manifest*, což jsou **metadata o assembly**. Manifest vždy obsahuje kompletní jméno assembly (neodpovídá jménu na disku) obsahuje textový název, číslo verze, kulturu a token obsahující veřejný klíč. Veřejný klíč je jednoznačný hash-kód přidělený assembly při kompilaci. Manifest dále obsahuje **metadata popisující obsah assembly**, tedy jaké jsou uvnitř definované jmenné prostory, jaké třídy a parametry daných tříd. Jednotlivé assembly potřebné pro spuštění programu jsou potom dodány buď při instalaci programu do složky programu, nebo mohou být uloženy v tzv. *Global Assembly Cache* (GAC). GAC je vlastně speciální složka obsahující všechny do ní vložené assembly bez ohledu na verzi/kulturu či jiné vlastnosti. Výsledkem je, že na počítači mohou být assembly ve více různých verzích a přesto si každá aplikace dokáže určit svou potřebnou (nevzniká DLL-hell[18]). O volbu správné assembly pro spuštěnou aplikaci se stará virtuální stroj.

6.2.3 Knihovny, Base Class Library

Base Class Library (BCL), neboli knihovna základních tříd, definuje funkcionalitu poskytnutou všem jazykům postaveným pro MS .NET Framework. BLC poskytuje třídy zastřešující množství funkčností, od základních datových typů a operací s nimi přes čtení a zápis souborů, renderování grafiky. Na základu této knihovny jsou postaveny ostatní knihovny určené například pro práce s databází a XML daty, webové aplikace a další. Fyzicky se jedná o množinu assembly (tedy souborů) umístěných v GAC, tedy přístupné všem programům vyvinutým pro .NET. Většinou se k BCL ještě explicitně zmiňují knihovny pro ADO.NET – tedy knihovny pro přístup

[18] Problém, kdy se na počítači vyskytují stejně nazvané DLL knihovny v různých verzích a aplikace má problém identifikovat konkrétní knihovnu, kterou potřebuje ke svému běhu. Blíže např. viz http://en.wikipedia.org/wiki/DLL_Hell.

k datům (databáze, XML aj.), knihovny pro tvorbu desktopových Windows aplikací (WinForms) a knihovny pro tvorbu webových aplikací ASP.NET – Web services.

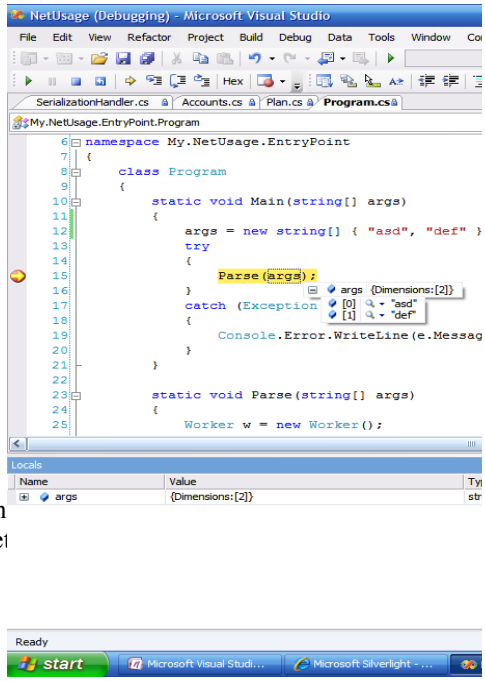
6.3 Jazyky

Všechny jazyky, pomocí kterých lze vytvářet .NET aplikace, jsou plně objektové. Navíc, díky překladu do mezikódu, mezi sebou mohou kooperovat a využívat tak knihovny napsané v jiných jazycích .NET.
Microsoft při uvolnění .NET Frameworku dodal pro vývoj 2 základní jazyky. Prvním je C# (čti sí šárp), který syntakticky vychází z C++ a Javy. Je primárním a zřejmě nejčastěji používaným jazykem pro práci v .NETu. Alternativním jazykem pro vývoj je pro někoho snáze čitelnější Visual Basic .NET (VB.NET). Přestože je často odsuzován, s předchozími verzemi VB má společnou pouze syntaxi, základy má postavené na CLI a z hlediska možností použitelnosti je s jazykem C# téměř identický. Jazyk C# bude více vyhovovat zkušenějším programátorům přecházejícím z C/C++ nebo Javy, jazyk VB.NET zase díky jinému (často přehlednějšímu) zápisu určitých funkčností bude vyhovovat začínajícím programátorům.
Dalším jazykem uvolněnými spolu s .NET Frameworkem byl C++/CLI[19]. Nicméně třetími stranami (specifikace CLI je otevřená) dochází ke vzniku nových jazyků jako odnoží stávajících. Za nejzajímavější lze jmenovat Boo (vychází z Pythonu), NetCOBOL (COBOL), Chrome (ObjectPascal), Delphi.NET, Haskel for .NET, IronLisp, P# (Prolog), Ruby.NET, #Smalltalk, J# (Java +, J++), F# (funkcionální programování), PHP.NET a další.
Nicméně, jak bylo zmíněno, všechny jazyky vycházejí z obecné specifikace a stejná funkčnost implementovaná v různých jazycích bude v MSIL vypadat stejně nebo velmi podobně. Jazyky navíc mezi sebou mohou kooperovat (po přeložení do MSIL).

6.4 Vývojové prostředí

Výchozím vývojovým prostředím pro tvorbu aplikací v .NET Frameworku je produkt nazvaný *Microsoft Visual Studio*. Podle vydaných verzí .NET Frameworku se přidává přípona (zpravidla podle roku vydání). Současnou verzí je Microsoft Visual Studio 2010 (MS VS 10) a připravuje se verze 2012.
Jedná se o moderní vývojové prostředí obsahující moderní prvky pro ulehčení zátěže programátora při psaní kódu. Samotný produkt vychází v několika tzv. edicích, které se navzájem liší dostupností jednotlivých funkcí. Tyto edice (názvy a schopnosti) se liší téměř u každé verze[20], typicky lze ale nalézt tři základní:

- MSVS Express – je na rozdíl od ostatních edicí **zcela zdarma**[21] a lze jej stáhnout ze stránek



[19] Ve skutečnosti existoval ještě derivát jazyka Java.
[20] Typicky podle změn názvů edic např. u uvolňovan
[21] A to i pro komerční účely – v této edici tedy může

výrobce. Jedná se o sadu odlehčených jednotlivých IDE určených především pro začátky. Pro každý jazyk existuje jedna varianta edice (nelze tedy použít více jazyků .NET), navíc jsou typicky instalované služby s IDE použitelné / dostupné pouze z daného počítače. Obsahuje pouze malé množství nástrojů a nemají úplnou nápovědu. Jednotlivé nástroje (k VS 2010) jsou Visual Basic Express, Visual C++ Express, Visual C# Express a Visual Web Developer Express.

- MSVS Professional – je již placenou edicí, typicky používanou v menších programátorských firmách. Podporuje integrované IDE pro všechny jazyky MS, celou nápovědu. Lze v ní vytvářet plnohodnotné aplikace s využitím různých průvodců a návrhářů (například i doplňků pro MS Office). Navíc obsahuje nástroje pro integraci Microsoft SQL Serveru, možnost vzdáleného ladění a další možnosti rozšíření (add-ons).
- MSVS Team System – je komplexní řešení pro vývoj aplikací ve velkých firmách. Obsahuje nástroje pro vývoj, spolupráci, metriky a reportování. Kvůli komplexnosti se dělí na edice Team Foundation (výchozí edice), Architecture Edition, Database Edition a Test Edition. Nejvyšší verze zahrnující všechny tyto edice se jmenuje Team Suite Edition.

Součástí vývojového prostředí Microsoft Visual Studio je i nápovědní systém MSDN. Ten je buď distribuován na DVD a instalován jako volitelná součást, nebo jej lze najít na internetu na adrese http://msdn.microsoft.com/library.

6.5 Základní práce v C#

Pro představu o technologii je třeba si udělat základní obrázek o možnostech programovacího jazyka. Jazyk C# je (například v porovnání s jazykem Java) velmi mocný a obsahuje nejrůznější jazykové konstrukce, které v konkurenčních jazycích vytvořit nelze. Protože je tento jazyk také nejrozšířenějším, bude představen více.
Jazyk C# (sí šárp) je zřejmě nejpoužívanějším jazykem pro vývoj aplikací pro .NET Framework. Jeho syntaxe vychází zčásti z C/C++ a z části z Javy, navíc má doplněny další konstrukce pro své specifické funkčnosti. Přestože zdrojový kód je opravdu velmi podobný, liší se v některých vlastnostech zásadně.
Pro další text předpokládáme seznámenou s problematikou programování a objektově orientovaným přístupem.

6.5.1 Hierarchie tříd

Jak bylo zmíněno, všechny jazyky, a tedy i C#, jsou **plně[22] objektové jazyky**. To znamená, že existuje třída *Object* a všechny ostatní třídy definované buď programátorem, nebo v BCL jsou přímým nebo nepřímým potomkem této třídy. Jednotlivé třídy jsou samozřejmě pod sebe řazeny hierarchicky, podle své funkčnosti.
Kromě hierarchie generalizace-specializace jsou jednotlivé třídy uspořádávány do celků podle své významové příslušnosti.

[22] „Plně“ znamená, že neexistuje typ, která by byl mimo hierarchii objektů. Například jazyk Java není plně objektový, protože primitivní typy nejsou v hierarchii objektů.

Jazyk samozřejmě podporuje základní principy objektově orientovaného paradigmatu, umožňuje jednoduchou dědičnost[23]. Steně jako jazyk C++ umožňuje pouze **explicitní**[24] virtualizaci.

6.5.2 Základní konstrukce

Objektové jazyky nabízejí několik konstrukcí, které lze při tvorbě objektů uplatnit. V následujícím přehledu bude vysvětlen význam nejzákladnějších z nich.

Rozhranní, třídy, struktury

Základní definice v jazyce C# je **(datový) typ**. Tyto typy jsou buď **referenční typy** (předávají se odkazem), nebo **hodnotové typy** (předávají se hodnotou).
Tzv. **třída** (pojem známý s jiných programovacích jazyků) je speciálním případem referenčního typu. Běžným pojmem je také **struktura**, což je speciálním případem hodnotového typu.
Třetí konstrukcí, která nevystihuje přímo funkčnost, ale používá se ke specializaci obecných požadavků na objekt, je **rozhranní** – interface. Jeho použití je běžné jako v obdobných programovacích jazycích.

Pole, Vlastnosti

Základními prvky, které se používají ve třídách pro udržení hodnoty, jsou pole[25] (field) a vlastnosti (properties).
Pole je konstrukce známá z obdobných programovacích jazyků – jedná se o běžnou instanční třídní proměnnou, která může nabývat hodnoty, a typicky je zapouzdřená.
Vlastnost je oproti tomu konstrukce převzatá z COM objektů. Jazyky, které vlastnosti nemají, je zpravidla nahrazují konstrukcí getXXX/setXXX. V C# se tedy gettery a settery nepoužívají.

Metody

Základním stavením prvkem pro vykonání funkčnosti v objektu je metoda. Pojetí metod se oproti ostatním jazykům C# nijak zásadně neliší.

Delegáti, události

Složitější konstrukcí jsou potom události a delegáti. Jazyk C# podporuje událostní programování – objekt tedy může sám aktivně informovat své okolí o nějakém ději nebo změně stavu[26]. Bližší výklad přesahuje rozsah studijní opory, proto pouze stručně: typ může definovat událost, kterou chce vyvolávat (například síťové připojení může vyvolávat událost „přišla data“). Delegát je „ukazatel“ na metodu, která bude danou událost obsluhovat (tedy metodu, která se zavolá v případě, kdy přijdou data).

---

[23] Nelze dědit z více předků, pouze jednoho.
[24] To znamená, že pokud chce programátor, aby se metoda chovala virtuálně, musí to explicitně naznačit pomocí klíčového slova. Implicitně virtuální je například jazyk Java, kde je každá metoda virtuální automaticky.
[25] Nezaměňovat s polem ve významu „array“!
[26] Jazyky, které toto neumožňují, tento princip nahrazují návrhovým vzorem *Observer*.

6.5.3 Další vlastnosti

Další vlastnosti budou představeny pouze stručně, pro srovnání s jinými jazyky. Celá platforma jazyků .NET umožňuje využívat reflexi (tedy analýzu zdrojového kódu za běhu). Jazyky běžně podporují generické typy.

6.6 Technologie použitelné v rámci .NET

Následuje seznam základních knihoven a technologií, které se běžně využívají při tvorbě aplikací.

6.6.1 ADO.NET + XML, LINQ

Základní knihovnou nad BCL je právě knihovna ADO.NET + XML. Zkratka ADO znamená „Active Data Object" a odkazuje se na dřívější technologii MS, která slouží k přístupu k datovým zdrojům různého typu – tedy různým databázím (typicky relačním), ale i XML souborům, a datovým souborům jiných formátů. U databází existují spolehlivé a funkční řešení pro všechny velké dodavatele databázových strojů.
Od verze FW 3.5 umožňuje jazyk C# provádět dotazy pomocí jazyka LINQ – „Language Integrated Query". Pomocí něj se lze **typově bezpečně** dotazovat nad databází – umožňuje tak eliminovat spoustu chyb při návrhu SQL dotazů, řešit problematiku persistence a ulehčit vývojáři tvorbu aplikací pracujících s databázemi. Výhodou je, že syntax jazyka LINQ lze použít na libovolné zdroje stejně (dotazy tedy budou vypadat stejně jak do databáze, tak do kolekce dat, nebo do xml souboru aj).

6.6.2 WinForms

WinForms je základní knihovnou určenou pro tvorbu destkopových aplikací operačního systému Windows. Lze říci, že pro tvorbu takových aplikací v současné době neexistuje lepší řešení. Pracuje na principu dědičnosti a událostním modelu.

6.6.3 ASP.NET

ASP.Net je spíše než knihovna soubor technologií pro tvorbu a běh webových aplikací. Jedná se základní a hlavní technologii pro web, ze kterého se následně odvíjejí deriváty podporující například návrhový model MVC.
ASP.NET je součást .NET Frameworku. Využívá tedy jeho CLR, čímž je zajištěno, že v ASP.NET lze psát zdrojový kód v libovolném jazyce, který implementuje CLS (Microsoft naimplementoval pro vývoj webových aplikací jazyky C# a VB.NET) a zvládá svou kompilaci pro ASP.NET.
Základním stavebním blokem je stránka. V ASP.NET lze HTML kód stránky a výkonný kód stránky oddělit do dvou souborů; první soubor obsahuje čistý kód HTML s prvky (+ drobná režie), druhý soubor obsahuje definici třídy související se stránkou a její funkčnost (ASP.NET je postaveno nad CLS, takže i webová stránka, stejně jako windows formulář, je třídou). Zrojový kód stránky může samozřejmě využívat libovolné, programátorovi dříve známé a dostupné knihovny a jejich datový typy z .NET frameworku.
Pro běh webové aplikace na počítači je třeba mít nainstalovanou nějaký internetový aplikační server. S ASP.NET se typicky váže produkt *Internet*

Information Services (IIS), který umožňuje běh aplikací v prostředí ASP.NET. IIS je běžných stanicích (edice OS Professional a vyšší) zpravidla volitelnou součástí instalace.
Technologie se přizpůsobuje a umožňuje využívat i nejmodernější prvky webu, jako jsou webové služby, technologie AJAX, HTML5 a další.
Nejdůležitější výhodou technologie ASP.NET však je, že princip tvorby webové stránky je téměř shodný s principem tvorby destkopové aplikace a díky tomu má technologie ASP.NET velmi vysokou učící křivku (řádově dny).

6.6.4 .NET Remoting

.NET Remoting je původní technologie .NET pro komunikaci mezi vzdálenými počítači. Funguje na „nízké“ úrovni na principu otevírání a zasílání dat po kanálech přes protokoly TCP/UDP. Její nevýhodou je svázanost technologií – aplikace obou komunikujících počítačů musí být napsané v .NET využívající .NET Remoting. Dnes je tato technologie téměře nevyužita a nahrazena technologií WCF.

6.6.5 WCF – Windows Communication Foundation

V současné době se rozmáhá technologie Web Services, která umožňuje agentům zasílat požadavky na vykonání operací na jiném počítači, bez ohledu na komunikující technologie. Technologie webových služeb je představena i v této studijní opoře.
Technologie .NET podporuje práci s webovými službami pomocí blokuWCF a pomocí různých průvodců umožňuje programátorovi snadno a jednoduše realizovat bloky aplikace fungující jako servery i klienti technologie web services.

6.6.6 WPF – Windows Presentation Foundation

Microsoft uvolnil tuto technologii pro tvorbu uživatelského rozhraní založenou na derivátu jazyka XML. Tento jazyk nazval XAML[27] a umožňuje definovat uživatelské rozhraní jako jednoduchý XML soubor. Výhodou je možnost použití tohoto řešení v rozličných zařízeních – s takovým souborem si poradí i ta nejjednodušší zařízení (viz .NET Micro Framework), ale lze je použít i pro desktopové aplikace nebo pro webové aplikace (viz Silverlight), přičemž syntax jazyka zůstává zcela shodná bez ohledu na cílové zařízení. Využívá opět událostní programování pro ovládání svých prvků.

6.6.7 Silverlight

Silverlight je dnes již známá technologie určená pro tvorbu dynamických webových stránek. Je tedy také součástí technologie .NET a její výhodou je založení na technologii WPF a využití knihoven .NET, díky čemuž opět i neznalý programátor Silverlightu (ale seznámený s .NET) dokáže rychle vytvářet odpovídající aplikace.

[27] Čti „zaml“.

6.6.8 ASP.NET MVC

Jenom stručně, tato odnož ASP.NET umožňuje využívat návrhového vzoru/architektury Model-View-Controller i v prostředí ASP.NET a nahrazuje tak klasický událostní model.

6.6.9 XNA

XNA je další z technologií postavených nad .NET a slouží k vývoji her a grafických aplikací v DirectX nad platformami MS Windows, Windows Phone 7, Xbox 360 a Zune. Základním principem je zapouzdření DirectX sadou tříd, které umožňují využívat funkce DirectX i méně zkušeným programátorům.

Kontrolní otázky:

1. Vyjmenujte, co se skrývá pod technologií .NET.
2. Objasněte, zda je možno vyvíjet nad technologií .NET zdarma.
3. Vysvětlete pojem ASP.NET a k čemu se vztahuje.
4. Vysvětlete pojem WCF a problematiku, kterou řeší v .NET.

Úkoly k zamyšlení:

V rámci této studijní opory jste ještě seznámeni s pojmem Webové služby. Zkuste se zamyslet, jaký je rozdíl mezi webovými službami a .NET remotinem a naopak, co mají tyto technologie společné. Která je výhodnější a proč?

Korespondenční úkol:

Pokuste se shrnout do tabulky výhody a nevýhody platformy .NET vzhledem k jiným programovacím jazykům, které znáte.

Shrnutí obsahu kapitoly

Kapitola představila technologii .NET – byly vysvětleny cíle platformy, způsob implementace, verze, kterými při svém vývoji technologie prošla a vývojové nástroje. Byly představeny základní vlastnosti jazyků .NET a zkráceně popsán způsob tvorby aplikací v .NET. Navíc byly představeny technologie, které lze spolu s platformou .NET využít.

7 Java

V této kapitole se dozvíte:

- Jaké jsou edice platformy Java a k čemu se používají.
- Jaké jsou klíčové technologie pro tvorbu webových aplikaci v Javě.
- Jaké jsou základní frameworky a možnosti automatizovaného testování při vývoji webových aplikací v Javě.

Po jejím prostudování byste měli být schopni:

- Orientovat se v edicích platformy Java.
- Orientovat se v architektuře popsaných frameworků a popsat jejich zásadní rozdíly.

Klíčová slova této kapitoly:

Java, nástroje, frameworky.

Doba potřebná ke studiu: 5 hodin

Průvodce studiem

Kapitola představuje platformu Java a rozebírá základní rozdíly edicí Javy. Důraz je kladen na Java Enterprise Edition a její klíčové technologie. V závěru jsou popsány frameworky, které ulehčují vývoj webových aplikací v Javě.
Na studium této části si vyhraďte 10 hodin.

Java vznikla v roce 1995 jako minimalistický programovací jazyk vyvinutý společností Sun Microsystems, jehož API obsahovalo pouze 211 tříd. Syntaxe jazyka vycházela ze syntaxe jazyka C/C++. Hned po uvedení sklidila velký úspěch a v dalších verzích se rychle zlepšovala. V roce 1999 se zásadně přepracovaly knihovny API, počet tříd vzrostlo na 1524 a Java i celá platforma byla označena jako Java 2. V současné době je aktuální verze Java 7 s 3977 veřejnými třídami. S vývojem samotného API bylo potřeba specifikovat i jeho použití. Proto dnes Java nepředstavuje pouze samotný programovací jazyk, ale celou platformu, kterou lze využít pro tvorbu programu pro přenosná zařízení, osobní PC, servery i jednoúčelové stroje v domácnosti (pračky, myčky).

Narozdíl od konkurenční platformy .NET je zde pouze jeden programovací jazyk. Obecně lze rozdělit jazyky na interpretované a kompilované. Mezi interpretované jazyky se řadí např. PHP či Visual Basic a provádění realizuje interpreter, který programový kód překládá řádek po řádku za běhu programu. Výhodou je to, že na jazyk nejsou kladeny vysoké nároky (netřeba inicializovat datové typy) a také je jednodušší nalezení chyby v kódu. Tato výhoda je vykoupena celkovou rychlostí zpracování.
Mezi kompilované jazyky lze zařadit C a C++. Narozdíl od interpretovaných jazyků probíhá překlad všech řádků kódu před spuštěním samotného programu. Kompilátor zkontroluje správnost syntaxe a vytvoří binární soubor v nativních instrukcích procesoru konkrétní platformy.

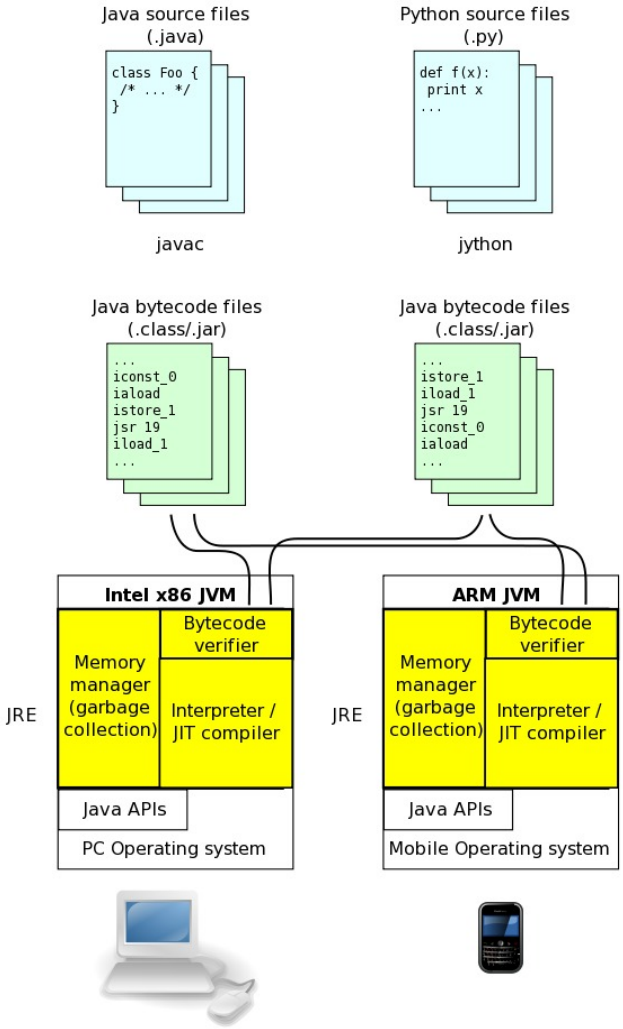
Jazyk Java se ovšem řadí do obou skupin. Nejdříve je zdrojový kód jazyka zkompilován do tzv. bajtového kódu (byte code) a ten je následně interpretován

Java virtuálním strojem (Java Virtual Machine). Virtuální stroj tvoří prostředníka mezi kódem jazyka Java a cílovou platformou (počítačem a specifickým OS). Při běhu se tedy vždy využíva JVM a proto se v některých zdrojích označuje Java jako interpretovaný jazyk. To ovšem není přesné, protože při kompilaci se nevytváří pouze bytový kód, ale dochází také k optimalizacím vlastního kódu (zjednodušení cyklů, cachování proměnných atd.).

Java se také často označuje za plně objektově-orientovaný programovací jazyk. Mezi plně objektově-orientované jazyky se řadí např. Simula či Beta. Java má ovšem definovány primitivní datové typy (byte, short, int, long), které jsou doménou jazyků procedurálních. Proto je vhodné označovat jazyk Java jako hybridní.

Vytvoření programu a jeho spuštění se tedy provádí v těchto krocích:

- Vytvoření programu v programovacím jazyku Java (soubory s příponou .java)
- Soubory se zkompilují překladačem do bytového kódu (přípona .class)
- Při spuštění přistupuje program k Java API virtuálního stroje
- Virtuální stroj Javy interpretuje program jako instrukce strojového kódu pro konkrétní platformu.



Obr. 7-1: Architektura Java Virtual Machine (Zdroj: Derrick Coetzee)

Vlastní kompilátor Javy prošel taktéž vývojem, od verze 7 lze zkompilovat do bytového kódu i jiné jazyky (např. Python). Při tvorbě platformy Java se tvůrce James Gosling držel základního pořadavku - multiplatformnost. Sun toto později označoval sloganem WORA (Write Once Run Anywhere), ktedy jednou napiš a spusť kdekoliv. Proto je také mezi kódem a cílovou platformou umístěn virtuální stroj zmíněný výše. Lze tedy říci, že kód java lze spustit na jakékoliv plaformě, pro kterou je vytvořen virtuální stroj. Pokud tedy chcete spustit svůj vlastní kód, musíte na cílový operační systém nainstalovat běhové prostředí - Java Runtime Environment (JRE), jehož součástí je i virtuální stroj. JRE dále obsahuje kromě stroje samotného i podpůrné knihovny, které využíva API.

Platforma Java se během vývoje rozdělila na tři hlavní edice:

- **Java ME (Micro Edition)** - vytvořena pro použití v přenosných zařízeních
- **Java SE (Standard Edition)** - standardní edice Javy pro použití s pracovními stanicemi (PC, Linux, Mac), obsahuje sadu pro tvorbu uživatelského rozhraní
- **Java EE (Enterprise Edition)** - je určena pro serverové nasezení, postavena na API Java SE a jsou přidány třídy pro podporu vývoje serverových řešení a webových aplikací(EJB, servlety, webové služby)

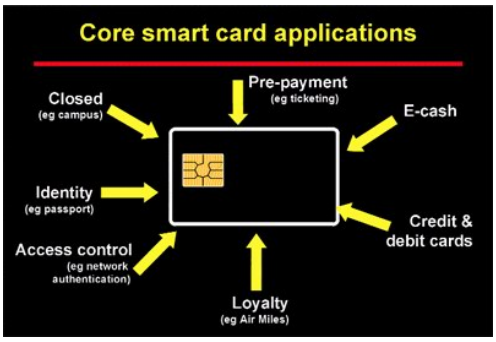
Speciální edici ještě tvoří Java Card, která se zpravidla řadí před Java ME a je určena pro vývoj jednoduchých jednoúčelových specifických aplikací (např. SIM karty GSM operátorů).

Každá z těchto edicí bude dále popsána. Oproti konkurenčním platformám má Java ještě jednu podstatnou výhodu - Java Community Process (JCP). Jedná se o mechanismus, který začleňuje zainteresované vývojáře přímo do vývoje platformy Java. Pokud zavítáte na stránky http://jcp.org, najde te zde plno standardů, které sami využíváte (např. JSR 221 - JDBC API Specifikace). Všechny tyto dokumenty byly postupně představeny programátorské veřejnosti a museli projít připomínkovacím a schovalovacím procesem. Lidé sami tak vytvářeji podobu nové verze Javy. JCP funguje od roku 1998 a v současné době obsahuje přes 380 specifikací. Hlavním výstupem jednoho procesu JCP je Java Specification Request (JSR) dokument. Jedná se o formální dokument, který navrhuje specifikace a technologie, které by měly být do Javy přidány. Součástí dokumentu je i referenční implementace. Dále budeme v textu odkazovat u popisovaných technologií také odkazy na jejic JSR dokument, aby si čtenář v případě hlubšího zájmu mohl dokument vyhledat.

7.1 Java Card

Edice Java Card byla představena v roce 1996 a byla určena pro bezpečné spouštění aplikací na zařízeních s minimem paměti. Pro samotné aplikace je využito technologie appletů a jako hardware jsou použity tzv. smart cards,

které najdete např. v SIM kartách nebo novějších platebních kartách. Java Card byl navržen tak, aby se Java applety byly přenositelné mezi kartami stejně jako standardní Java aplikace pro PC. Pro samotné spouštění aplikací je vytvořen speciální virtuální stroj (Java Card VM), který používá velmi podobný bitový kód jako standardní edice Javy.



Obr. 7-2: Applety ve Smart Card (Zdroj: http://www.expresscomputeronline.com/20030623/focus1.shtml)

Důraz u Java Card je kladen hlavně na bezpečnost a proto byly do standardu zapracovány tyto mechanismy:

- Applet firewall - Na jednom virtuálním stroji běží zpravidla více appletů. Java Card VM obsahuje mechanismy pro kontrolu přístupu k datové části jiných appletů.
- Šifrování - Je implementována podpora jak pro symetrické šifry (DES, 3DES, AES), tak asymetrické (RSA).
- Data Encapsulation - Každá aplikace (applet) je spuštěna v izolované prostředí nezávisle na operačním systému a hardware.

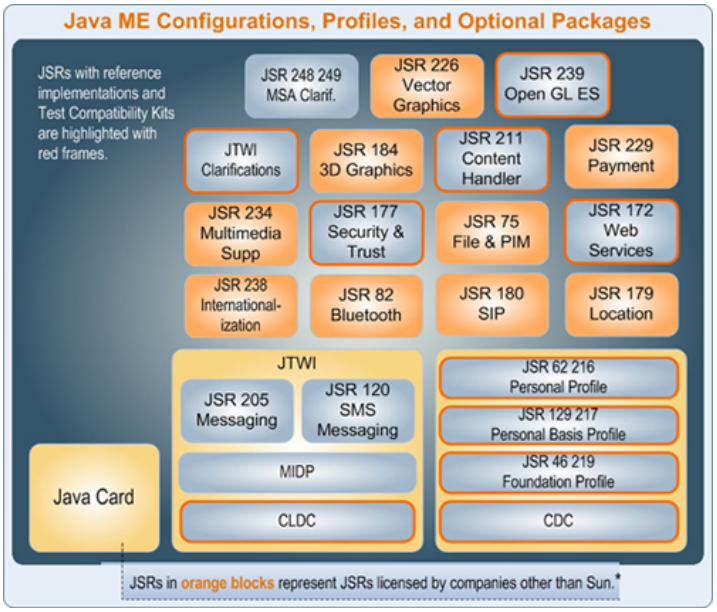
Z hlediska programovacího jazyku je Java Card podmnožinou jazyka Java. Jinými slovy, všechny konstrukce, které se použijí v Java Card, fungují ve standardní Javě.
V současnosti je aktuální verze 3, která rozděluje platformu na dvě edice:

- Classic Edition - Přirozený následník předchozího standardu.
- Connected Edition - obsahuje nový virtuální stroj a orientuje se více na síťové prostředí (podpora více vláken, možnost využít komunikačních schémat SOAP).

7.2 Java ME

Edice Java Micro Edition (ME) byla vytvořena pro přenosná zařízení s omezenými zdroji (výkon procesoru, paměť). Dříve bylo její hlavní použití v oblasti mobilních telefonů, ale nyní, kdy tři hlavní softwarové mobilní platformy(Android, iOS, Windows Phone) mají své vlastní SDK, se Java ME používá hlavně v set-top boxech a mobilních telefonech s proprietárním OS.

U mobilních zařízení je kromě problémů s nedostatkem hardwarových zdrojů také problém se zobrazením informace. Typů mobilních telefonů je velké množství a každý typ má jiné rozlišení obrazovky. Proto je Java rozdělena do několika profilů pro různá cílová zařízení.



Obr. 7-3: Konfigurace, profily a volitelné balíčky Java ME (Zdroj: http://developers.sun.com/mobility/getstart/)

Pro nás jsou v tuto chvíli důležité dvě základní konfigurace: CDC (Connected Device Configuration) a CLDC (Connected Limited Device Configuration).

7.2.1 CLDC konfigurace

Konfigurace CLDC (JSR 139) je určena pro malá zařízení, které mají následující minimální HW nároky na paměť:

- 160 kB stálé paměti (pro uchování dat) ,
- 32 kB "nestálé" paměti (k dispozici za běhu VM),

Pro spuštění aplikací je taktéž potřeba virtuální stroj. JVM, který se používá pro standardní verzi Javy je ovšem moc velký a proto společnost Sun (nyní Oracle) vytvořila derivát virtuálního stroje nazvaný K Virtual Machine (KVM).

API v této konfiguraci obsahují velmi omezenou podmnožinu standardní edice Javy (SE), jedná se o vybrané a trochu upravené třídy z knihoven:

- *java.lang*,
- *java.io*,
- *java.util*.

Konfigurace obsahuje novou knihovnu pro I/O operace, s třídami u nichž nebylo možno zachovat podobnost se standardní edicí, jedná se o knihovnu *javax.microedition.io*.

V rámci této konfigurace je definován pouze jeden profil zvaný MIDP profil. Tento profil upřesňuje CLDC konfiguraci pro použití na nejmenších zařízeních, převážně hromadně rozšířených (mobilní telefony). Jedná se o

nejvyužívanější profil, který navíc k HW specifikaci CLDC přidává další omezení:

- minimální velikost displeje 96 x 54,
- možnost ovládat zařízení klávesami nebo dotykem obrazovky,
- alespoň 8 kB stálé paměti pro ukládání dat aplikací.

Ke knihovnám CLDC konfigurace přidává:

- *javax.microedition.rms* – správa trvalých dat,
- *javax.microedition.midlet* – obsahuje třídu MIDlet, základ MIDP profilu
- *javax.microedition.io* – k CLDC přidává třídu HttpConnection,
- *javax.microedition.lcdui* – třídy pro tvorbu uživatelského rozhraní.

Téměř všechny mobilní telefony spadají do kategorie MIDP a využívají tedy MIDP profilu. Podle základní třídy profilu MIDP (třída MIDlet) říkáme aplikacím pro tuto kategorii **midlet**. Výjimkou z této kategorie jsou některé komunikátory se silnějším procesorem a větší pamětí (např. Nokia 9210).

7.2.2 CDC konfigurace

CDC (JSR 218) konfigurace je cílená na zařízení s 32bitovým procesorem a alespoň 512 kB ROM a 256 kB RAM. Virtuální stroj této konfigurace musí zvládat prakticky stejnou funkčnost jako ve SE, tudíž obsahuje klasický JVM a stejnou sadu API.

CDC je popsána referenční implementací CDC Reference Implementation 1.1 (JSR 218). CDC profilů existuje několik, stručně si popíšeme jen následující:

CDC profilů existuje několik a jejich přesné znění lze nalázt v JSR dokumentech. Mezi nejvýznamněj patří:

- **Foundation Profile** - Přidává knihovny, které v CDC nejsou obsaženy kromě java.beans, java.rmi a java.sql.
- **Personal Basis Profile** - Přidáva uživatelské rozhraní (UI), které je omezeno použitím pouze jednoho okna.
- **Personal Profile** - Oproti Personal Basis Profile přidáva knihovny AWT.

Obě konfigurace (CLDC a CDC) lze doplnit o volitelné balíčky. Volitelné ve smyslu, že tyto balíčky přidá výrobce hardwaru daného zařízení (např. JSR 82 - Bluetooth, JSR 184 - 3D Grafika atd.).

7.2.3 MIDlety

Jak jsme již zmínili výše, MIDlety jsou aplikace, převážně pro mobilní telefony, psané podle profilu MIDP konfigurace CLDC. Tyto aplikace umožňují pracovat s grafikou, ukládat záznamy apod. MIDP konfigurace byla představena již v roce 1999 a postupně se dopracovala k aktuální 3. verzi schválené procesem JCP v roce 2009.

Nejdříve začněme s grafikou a uživatelským rozhraním (UI). Profil obsahuje prostředky pro práci s nízkoúrovňovými komponentami. Ty kreslí přímo na display, umožňují zachytávat události od kláves. Obsahuje také prostředky pro práci s vysokoúrovňovými komponentami, které jsou nezávislé na typu a velikosti displeje. Pro práci s grafikou a UI používáme displej, který instancujeme jako singleton pomocí volání metody *Display.getDisplay()*. Pro zachycení událostí (stisknutí tlačítka mobilu) existuje *CommandListener*. Následující obrázek ukazuje grafické komponenty MIDletu.

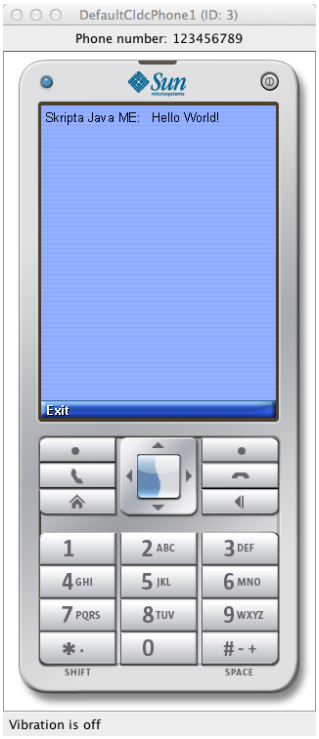
Pokud bychom chtěli vypsat na obrazovku mobilu "Skripta Java ME: Hello World!", vypadalo by to např. takto:

```
package hello;

import javax.microedition.lcdui.*;
import javax.microedition.midlet.MIDlet;

public class Midlet extends MIDlet implements CommandListener{
    // Displej
    private Display display;
    // Hlavni formular
    private Form form;
    // Zprava
    private StringItem stringItem;
    // Pro ukonceni aplikace
    private Command exitCommand;

    public void commandAction(Command command, Displayable displayable) {
        if (displayable == form) {
            if (command == exitCommand) {
                destroyApp(true);
            }
        }
    }

    public void startApp() {
        // Vytvor formular
        stringItem = new StringItem("Skripta Java ME:", "Hello World!");
        form = new Form(null, new Item[] {stringItem});
        exitCommand = new Command("Exit", Command.EXIT, 1);
        form.addCommand(exitCommand);
        form.setCommandListener(this);
        // Ziskej displej pro kresleni - Singleton!
        display = Display.getDisplay(this);
        display.setCurrent(form);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
        display.setCurrent(null);
        notifyDestroyed();
    }

}
```

Obr. 7-4: Midlet - výpis na display

Využíváme zde dvou importů - *lcdui* pro práci s displejem zařízení a *MIDlet* pro základní strukturu midletu. Navíc implementujeme dříve zmíněný zachytávač událostí - *CommandListener*. Metoda startApp je volána při spuštění midletu v mobilním zařízení. Zde se nejprve vytvoří objekt pro testovací řetězec, poté formulář pro zobrazení řetězce na displeji a nakonec se metodou *getDisplay* získá odkaz na displej zařízení. Poté se pouze formulář pošle na displej. Abychom aplikaci mohli korektně ukončit, musíme přidat také potřebný ovládací prvek. Proto je nejprve vytvořen příkaz *exitCommand*, který se přidá do formuláře a sváže se s zachytávačem událostí (metoda

setCommandListener). Abychom aplikaci otestovali, nemusíme ji zkompilovat a nahrát do konkrétního mobilu. Vývojové prostředí nabízejí své vlastní emulátory, které výsledek věrně zobrazí. Výsledek výše popsaného kódu tak může vypadat ve vývojovém prostředí NetBeans takto:



Obr. 7-5: Midlet v emulátoru

7.2.4 Úložiště záznamů

Stejně jako v rozsáhlých podnikových aplikacích, potřebujeme i v mobilních aplikacích ukládat data pro pozdější použití. K tomuto slouží v mikro verzi Javy tzv. Record store. Jedná se o seznam záznamů, které se skládají z pole bajtů (záznamy record store jsou indexovány od 1). Můžeme tedy říci, že se jedná o velmi jednoduchou databázi. Jednotlivé databáze musí mít v rámci jedné sady midletů unikátní názvy. Za konzistenci databáze při práci s ní odpovídá implementace Javy (operace nad databází jsou atomické, nelze provádět transakce). Pokud odstraníme sadu midletů z telefonu, jsou automaticky odstraněny i všechny její databáze. Pro práci se záznamy využijeme balík *javax.microedition.rms*.

7.3 Java SE

Java Standard Edition (SE) je cílena pro použití na destopových zařízení. Z hlediska vývoje informačních systému se jedná klasickou aplikaci, nebo v případě aplikace distribuované o tzv. "tlustého klienta". Základy programování Javy jste získali v dřívějším studiu. Zde se tedy omezíme hlavně na funkce podporující distribuovanou architekturu a některé speciální techniky. Java SE obsahuje kompletní API, není omezeno z hlediska počtu tříd. Naopak je omezeno virtuálním strojem. Nejprve musí pro danou platformu virtuální stoj existovat. Pokud existuje, musí být také kvůli bezpečnosti aktualizován.

Některé operační systémy obsahují virtuální stroj Javy přímo v instalaci samotného OS, jinde je třeba stáhnout dodatečnou instalaci ze stránek společnosti Oracle. Pro produkční systémy postačuje Java Runtime Environment (JRE), pokud potřebujete na daném stroji i vyvíjet (potřebujete kompilátor jazyka Java), musíte nainstalovat verzi Java Development Kit (JDK).

7.3.1 Java a komunikace

Při vývoji jazyka Java bylo od začátku myšleno na použití jazyka v distribuovaném prostředí. Nejjednodušší způsob pro vytvoření aplikace s architekturou klient-server je využít soketů (balíček *java.net.**). Sokety jsou svým způsobem primitivní formou komunikace. Sice programátora odstiňují od od navazování spojení, samotného kódování přenosu, ale nenabízejí žádný komfort při programování. Programátor tak zpravidla vytváří svůj vlastní komunikační protokol a transakční zpracování. Sokety přímo nedovolují vzdáleně volat procedury vytvořených tříd, slouží pouze jako jednoduchý komunikační most pro přenost informací ve formě objektu (po serializaci). Příklad použití soketů by mohl vypadat například takto:

```
public class SocketServer {
    static boolean ServerNasloucha = true;
    //Spusteni serveru
    public static void main(String[] args) throws IOException {
        SocketServer srv = new SocketServer();
        srv.naslouchej();
    }
    //Metoda pro naslouchani klientovi
    public int naslouchej() throws IOException {
        ServerSocket serverSocket = null;
        int cisloPortu = 8085;
        //Snazi se otevrit socket na kterem bude server poslouchat
        try {
            serverSocket = new ServerSocket(cisloPortu);
        } catch (IOException e) {
            System.exit(1);
        }
        while (ServerNasloucha) {
            SocketServerThread obsluhaKlienta;
            // poslouchа na portu 8085
            obsluhaKlienta = new SocketServerThread(serverSocket.accept());
            // obsluhuje klienta
            obsluhaKlienta.start();
        }
        serverSocket.close();
        return 0;
    }
}
```

Obr. 7-6: Použití soketů - serverová část kódu

Na straně serveru programátor vytvoří metoda, která obsahuje nekonečný cyklus. Po spuštění serverové části si JVM rezervuje port (v tomto případě 8085) a očekává na něm klienta. Pokud se klient připojí, je vytvořeno nové vlákno, které klienta obslouží. V této třídě se využívá třída *ServerSocket*, která je součástí balíčku *java.net*.

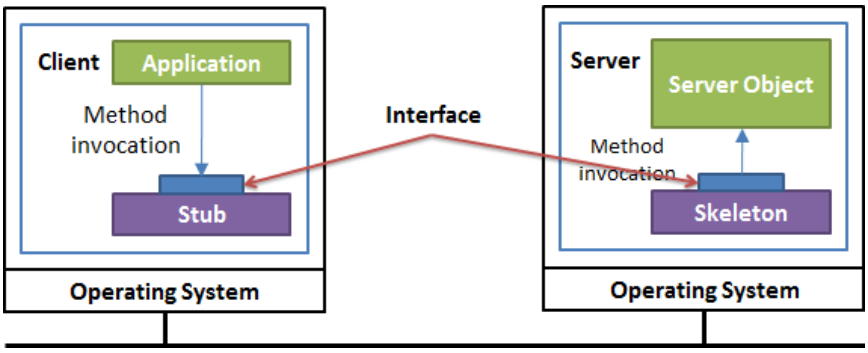
```
public class Client {
    private Socket clientSocket;
    private ObjectOutputStream oOut;
    private ObjectInputStream oIn;

    public void otevriSpojeni(String adresaServeru, int cisloPortu) throws UnknownHostException, IOException {
        try {
            // otevre spojeni na server
            clientSocket = new Socket(adresaServeru, cisloPortu);
            // pro posilani na server
            oOut = new ObjectOutputStream(clientSocket.getOutputStream());
            // pro precteni ze serveru
            oIn = new ObjectInputStream(clientSocket.getInputStream());
        } catch (java.net.UnknownHostException e) {
            // server nenalezen
            throw (e);
        } catch (java.io.IOException e) {
            throw (e);
        }
    }

    public void nactiRetezecZeServeru(String retezec) throws IOException {
        // --- Get the Response from the Server ---
        try {
            retezec = ((String) oIn.readObject());
        } catch (ClassNotFoundException e) {
            // server neodpovedel
        }
        System.out.println("Odpoved serveru: " + retezec);
    }
```

Obr. 7-7: Použití soketů - klient

Klientská část pak obsahuje metodu, která se připojí na server a přečte data (v tomto případě řetězec reprezentovaný objektem String). Všimněte si, že není žádným způsobem řešeno zabezpečení přenosu, to musí programátor také vyřešit sám.

Jak je z popisu patrné, sockety nejsou zcela vhodné pro každodenní práci. V objektově-orientovaném prostředí je vhodné pracovat přímo s objekty a jejich metodami. Proto byly vytvořeny knihovny Java Remote Invocation (RMI) pro realizování vzdáleného volání procedur. Samotná myšlenka není novinkou Javy, vychází z obecného konceptu Remote Procedure Call (RPC). RMI je tedy konkrétní implementací tohoto konceptu pro platformu Java. Původně bylo vyvinuto s vlastním protokolem Java Remote Method Protocol (JRMP) a bylo použitelné pouze pro komunikaci mezi dvěma virtuálními stroji Javy. Tato nepřijemnost byla později odstraněna implementací RMI-IIOP a tím zpřístupnění standardu Common Object Request Broker Architecture (CORBA) vytvořeném konsorciem Object Management Group (OMG). Je tedy možné vyvolat vzdálené volání objektu např. z platformy .NET.

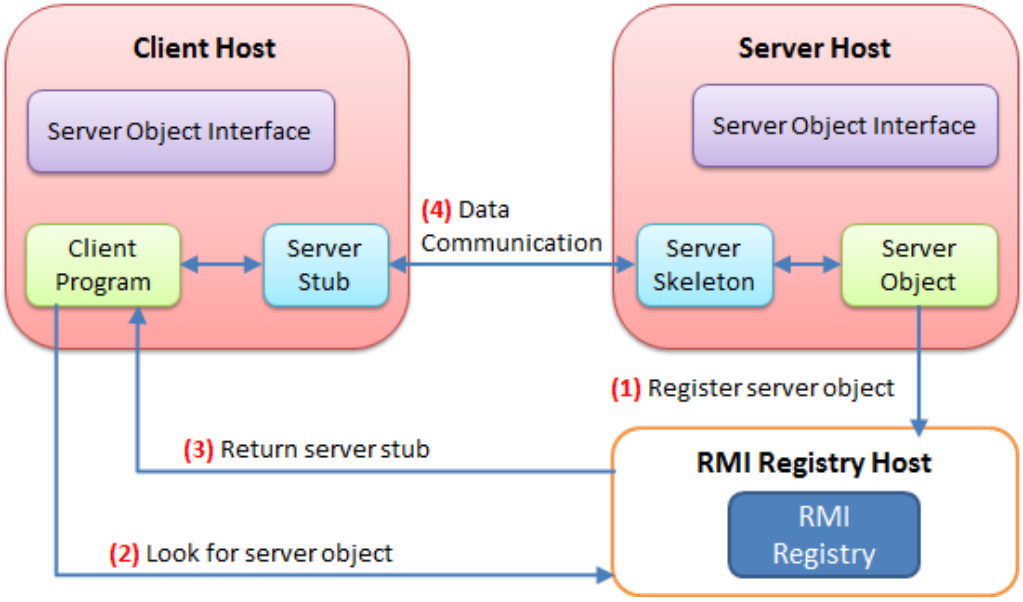
Architektura RMI je vcelku jednoduchá a obsahuje pouze dva důležité prvky - Stub a Skeleton.



Obr. 7-8: Stub a Skeleton (Zdroj: http://lycog.com/distributed-systems/stub-skeleton/)

Stub je přítomen na straně klienta a Skeleton na straně serveru (počítač, který poskytuje službu/objekt). V předchozích verzích Javy bylo potřeba vytvořit zvláštní třídu (soubor), který reprezentoval Stub či Skeleton. Od verze 1.6 již není potřeba vytvářet samotné soubory, identifikace a volání tříd probíhá pomocí reflexe. Aby bylo možno data přenést přes sít (teoreticky jakékoliv médium vč. flash disku) vykoná se proces označený jako *Marshalling*. Marshalling je do jisté míry synonymum pro serializaci, kterou znáte např. z persistence objektů. Hlavním rozdílem mezi Marshallingem a serializací je ten, že první jmenovaný je použit i pro přesun parametrů, kdežto serializace se orientuje hlavně na přenos dat. Lze také říci, že Marshalling lze provést serializací.

Pro komunikaci mezi klientem a serverem se využívá na nižší vrstvě soketů. Vzdálený objekt ovšem není definován pouze IP adresou a portem jako v případě socketů. Pracuje se zde s "obecnou" adresou - např. rmi://test.osu.cz/RMITest - která se skládá z adresy na RMI registr a jména objektu (v tomto případě RMITest). Po získání odkazu vzdáleného rozhraní již voláme konkrétní metody vzdáleného objektu stejně, jako metody objektu lokálního. Java RMI se postará jak o komunikaci, tak o Marshalling samotného volání.



Obr. 7-9: RMI ve čtyřech krocích (Zdroj: http://lycog.com/distributed-systems/

Volání pomocí RMI se tedy dá shrnout do čtyř kroků:

1) Registrace objektu v RMI registru.
2) Nalezení vzdáleného objektu klientem.
3) Vytvoření Stubu pro Marshalling.
4) Vlastní datová komunikace.

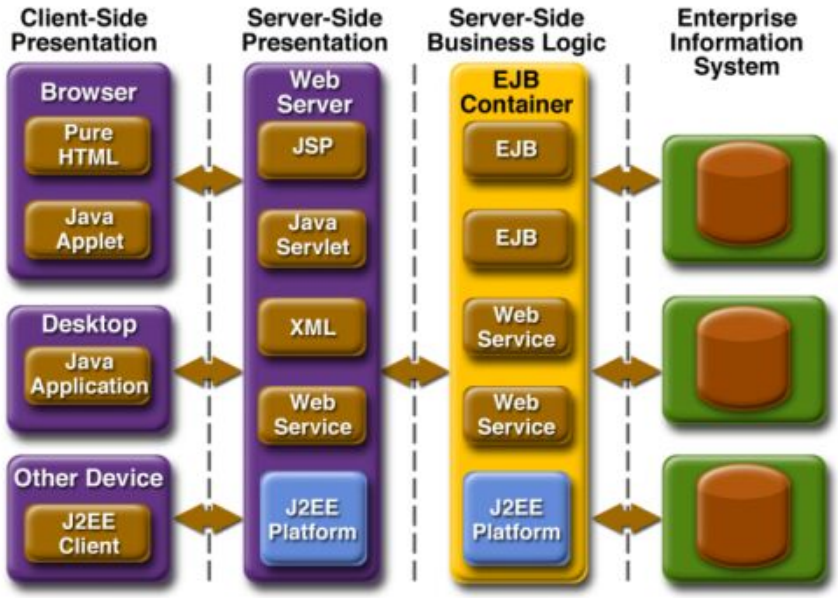
V současnosti nejvíce používanou a podporovanou technologií pro komunikaci mezi různými softwarovými platformami jsou webové služby. Principům fungování webových služeb se věnuje celá kapitola 8, zde jen připomeneme, že je možno je použít i u tzv. tlustých klientů v architektuře klient-server a také

pro komunikaci mezi jinými technologickými platformami (.NET, C++). Pro výměnu dat se ve webových službách využívá XML, ovšem pro komunikaci je potřeba více, než pouze výměna XML dokumentu. V API je proto vytvořen balíček *javax.xml.ws*, který zprostředkuje třídy potřebné pro komunikaci pomocí webových služeb. Od verze Java 5 lze pracovat s webovými službami velmi pohodlně pomocí anotací. Také je potřeba zdůraznit, že pro vytvoření serveru s webovou službou samotná Java SE nestačí a je potřeba využít minimálně platformu Java EE. Pro zjednodušení implementace webových služeb se často využívá již vytvořených frameworků, např. Apache Axis.

7.4 Java Enterprise Edition

Java Enterprise Edition (Java EE) je významným příspěvkem celé platformy v oblasti distribovaných řešení. Koncept začal vznikat pod sloganem "Write Once-Deploy Anywhere" (WODA) a je parafrází původního WORA. Autorům tedy šlo o to napsat jednu webovou aplikaci, která bude spustitelná kdekoliv. Jak si později vysvětlíme, není přenos aplikací úplně bez komplikací, ale v zásadě jsou aplikace přenositelné mezi různými servery. Při vývoji Javy EE bylo dále myšleno na škálovatelnost, výkonnost a bezpečnost spouštěných aplikací. V Jave SE se pro spuštění aplikací využívá JVM, pro platformy Java EE je situace komplikovanější. Nejprve se nainstaluje Java virtuální stroj, poté se spustí aplikace označovaná jako aplikační server a v rámci aplikačního serveru jsou umístěny kontejnery, které spouští webové aplikace. Samotná Java je tak rozšířením Javy SE o knihovny a technologie, které podporují výše zmíněné požadavky.

Následující obrázek ukazuje různé komponenty Javy EE a jejich rozdělení do logických vrstev aplikací.



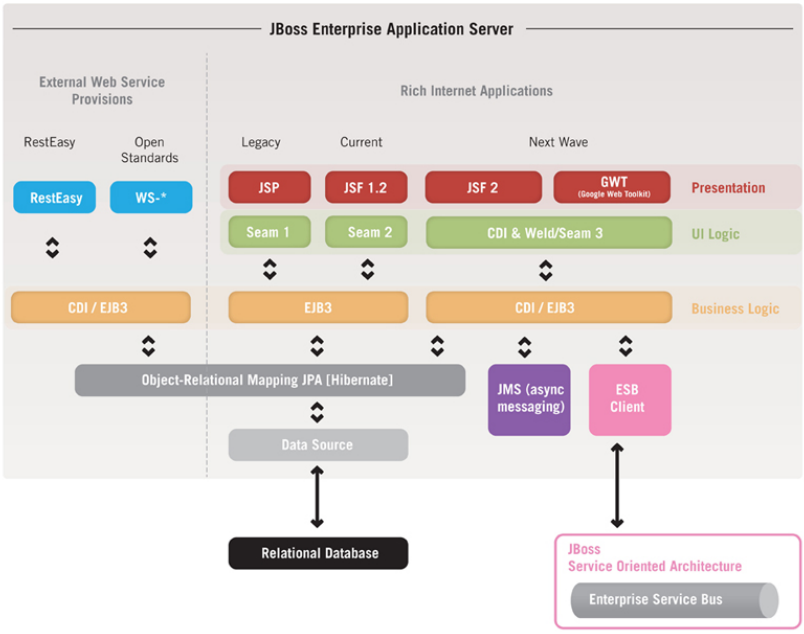
Obr. 7-10: Java EE komponenty v logických vrstvách aplikací (Zdroj: Suresh Devang)

Zleva je přítomna klientská část. Většinou se jedná o webový prohlížeč případně Java Applet. Klientská část přistupuje k prezentační logice serveru, ta

komunikuje s aplikační logikou a pro aplikační logiku jsou data získávána z databáze. Toto rozdělení je pouze logické, většinou je prezentační logika a aplikační (business) logika umístěna na jednom serveru. Hlavní technologie, které nás na platformě Java EE budou zajímat jsou *servlet*, *JavaServer Pages (JSP) a Enterprise JavaBeans (EJB).*

Aplikačních serverů je více a aby byla zaručena funkčnost aplikací, procházejí servery přísnou certifikací. Aktuální list certifikovaných aplikačních serverů lze nalézt na stránkách společnosti Oracle. Mezi nejpoužívanější komerční servery patří WebSphere od společnosti IBM. Samotná licenční politika IBM ovšem není vhodně nastavena vzhledem k finančním možnostem studentů a proto doporučujeme velmi kvalitní neplacené alternativy - JBoss, GlassFish a WebLogic.

Následující obrázek znázorňuje architekturu aplikačního serveru JBoss. Povšimněte si, že kopíruje logické rozdělení Java EE komponent z předchozího obrázku.



Obr. 7-11: Architektura aplikačního serveru (Zdroj: http://www.advancedcomputersoftware.com)

Technologie servlet a EJB potřebují vlastní běhové prostředí. Toto běhové prostředí se nazývá kontejner. Některé aplikační servery nabízejí pouze servletový kontejner (např. Apache Tomcat), jiné aplikační servery v sobě integrují více kontejnerů. Logicky servery, které podporují více technologií platformy Java EE, bývají velmi náročně na operační paměť a jejich start trvá podstatně déle.

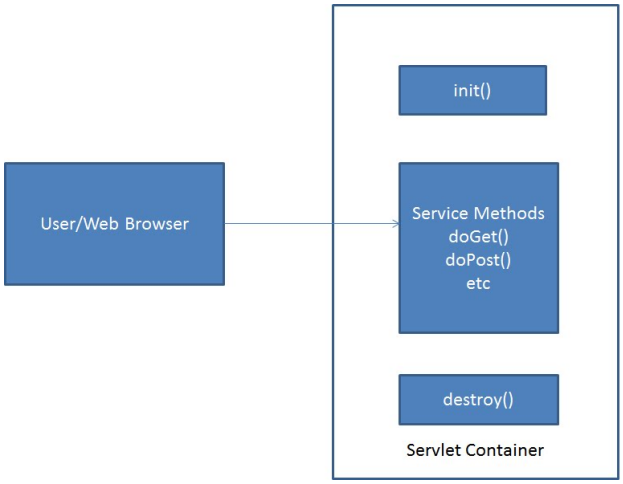
7.5 Servlety

Základní technologií v platformě Java EE jsou servlety (specifikace JSR-154). Původně byly servlety vyvinuty jako odpověď Javy na CGI skripty. Dneska jsou používány jako API pro tvorbu dynamických webových aplikací, ale také

jako integrační nástroj pro jiné technologie (webové služby, JSF). Aktuální verze je 3.0 a třídy jsou umístěny v balíčku *javax.servlet*.

Servlet je tedy nástroj pro práci s požadavkem klienta (HTTP request) a pro generování odpovědi (HTTP response). Jako výstup lze generovat jakýkoliv dokument (nejčastěji HTML a XML).

Princip fungování servletu lze vysvětlit na jeho životním cyklu:



Obr. 7-12: Životní cyklus servletu (Zdroj: http://inheritingjava.blogspot.cz)

Při spuštění aplikačního serveru se zároveň vytvoří servletový kontejner. Podle nastavení kontejneru se buď inicializují všechny servlety, nebo se čeká, který servlet bude zavolán jako první. Inicializací servletu se rozumí zavolání metody *init()*, kterou každý servlet musí obsahovat (i když je prázdná a nic nevykonává). Po inicializaci zůstává servlet v paměti a všechny příchozí požadavky obsluhuje tak, že vytvoří nové vlákno (nikoli proces). Během obsaluhy se používají příslušné metody podle požadavku. Pokud klient odesílá na server formulář metodou GET, volá se příslušná metoda - *doGet()*. Po skončení práce se servletem se zavolá metoda *destroy()*, která odstraní instanci ze servletového kontejneru. V praxi by se metoda *destroy()* mohla volat např. při odhlášení uživatele ze systému (nikoli zavření prohlížeče) a přidali bychom do ní smazání cookies nebo ukončení spojení s databází. Pokud z nějakého důvodu servletový kontejner havaruje, metoda *destroy()* se nevolá automaticky, ale servlet zůstává v paměti.

Následující kód ukazuje jednoduchou implementaci servletu:

```java
package cz.osu.infs2.servlets;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public void init(ServletConfig config) throws ServletException {
        // priprava databaze
    }

    public void destroy() {
        // ukoncovani spojeni, cisteni vyrovnavacich pameti
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        String hodnota = request.getParameter("text");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>"+hodnota+"</h1>");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

Obr. 7-13: Ukázka servletu

Ukázkový servlet obsahuje obslužné metody *init* a *destroy* a také metodu *doGet*, která se spouští při požadavku HTTP request metodou GET. Pokud tedy odešleme formulář vytvořený HTML stránkou na tento servlet, první řádek přečte hodnotu parametru "text" a jako výsledek uživateli pošle HTML stránku s vypsanou hodnotou tohoto parametru. Servlet sice programátor do jisté míry odstíní od protokolu HTTP, ale dokument HTML musí být generován ručně. Proto se od generování stránek pomocí servletů rychle upustilo a využívají se pro to technologie jiné, např. JavaServer Pages (JSP).

Protože HTTP protokol obsahuje více metod, je pro každou metodu v servlet API přítomna obslužná metoda (celkem 7 metod). Mezi základní patří:

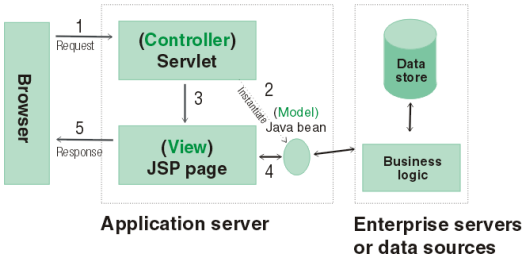
- GET - obsluhováno metodou doGet, přistupuje k parametrům v URL adrese.
- HEAD - obsluhováno metodou doHead, získává informace z hlavičky HTML dokumentu (např. typ dokumentu, velikost, kódování).
- POST - obsluhováno metodou doPost, slouží pro předávání parametrů mimo URL, zpravidla se využívá u rozsáhlých dokumentů a při uploadu malých souborů, teoreticky velikost parametru není omezena, prakticky ji ovšem omezuje konkrétní webový server délkou zpracovávání požadavku.

Samotné servlety se dnes již nepoužívají pro tvorbu webových aplikací. Své místo ale mají v integraci webových frameworků a jiných technologií. Servlet je totiž univerzální zpracovatel HTTP dotazů a odpovědí a tak jej využívají téměř všechny webové frameworky (bude vysvětleno dále). Servlet se také používá pro zpřístupnění techniky AJAX do stávající webové architektury.

7.6 Prezentační logika (JSP, JSTL)

Předchozí příklad generování HTML stránky pomocí servlet dobře demonstroval pracnost vytváření prezentační logiky pomocí servletů. Kombinování aplikační a prezentační logiky mělo za důsledek tvorbu nepřehledných a těžko udržitelných webových aplikací. Proto byl vytvořen

standard JavaServer Pages (pod specifikací JSR 245 ve verzi 2.1). Ten odděluje aplikační logiku od logiky prezentační. Architekturu zobrazuje následující obrázek:



Obr. 7-14: Architektura JSP (Zdroj: IBM)

Uživatel pošle HTTP dotaz na servlet, ten vytvoří instanci Java Beanu (pozor, neplést si s Enterprise JavaBean, to je jiná technologie). Bean pracuje jako model pro JSP stránku. Po vytvoření modelu předá servlet řízení JSP stránce, která přistupuje k Java beanu a vykresluje HTML stránku, kterou posílá uživateli. JSP není nutně omezeno na generování HTML dokumentů, může generovat např. XML. Jednoduchá JSP stránka a obslužný Java Bean tvořící model je uvedena na následujícímo obrázku:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<jsp:useBean id="beanNaStrance" scope="session" class="cz.osu.infs2.beans.MujBean" />
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Vítejte na INFS2</title>
</head>
<body>
<h1>Vítejte uživateli <jsp:getProperty name="beanNaStrance" property="username" />!
</h1>
</body>
</html>
```

```
package cz.osu.infs2.beans;

public class MujBean {
    private String username;

    public MujBean() {
        username = null;
    }

    public void setUsername(String name) {
        username = name;
    }

    public String getUsername() {
        return username;
    }
}
```

Obr. 7-15: JSP stránka a obslužný Java Bean

JSP stránka se skládá ze standardních HTML značkovacích tagů a speciálních tagů. Ty dovolují použití jazyka Java přímo ve stránce. Mezi základní se řádí direktivy, deklarace, výrazy a akce. K těmto základním se později přidaly speciální párové tagy <*jsp*>, které nahrazují původní a zpřehledňují zobrazení stránky pro programátora. Teoreticky lze totiž přidat celou aplikační logiku do JSP stránky bez rozdělení architektury, toto se ovšem nedoporučuje z důvodu přehlednosti a údržby kódu a proto se speciální tagy snaží nutit vývojáře k důslednému rozdělení architektury.

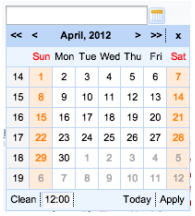
JSP stránky obsahují programový kód a proto se před umístěním na server stejně jako servlety musí kompilovat.

Podobně jako u servletů se v technologii JSP ukázalo, že ve složitějších aplikacích se určité problémy (řádky kódu) neustále opakují a bylo by je vhodné sjednotit. Pro tyto účely byl vytvořena specifikace JavaServer Pages Standard Tag Library (JSTL - JSR 52), která nabízí některé základní programátorské techniky (např. iterace) a také zjednodušují překlad implementací podpory standardu i18n.

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
.
.
.
<c:forEach var="person" items="${requestScope.list}">
                <tr>
                    <td><c:out value="${person.firstName}"  /></td>
                    <td><c:out value="${person.lastName}" /></td>
                </tr>
</c:forEach>
```

Obr. 7-16: Iterace pomocí JSTL

Na výše uvedeném příkladu je ukázáno procházení listu pomocí JSTL. Abychom mohli knihovnu tagů použít, je nutno i importovat a definovat jí vlastní prefix (v tomto případě "c"). Poté již v kódu využijeme párový tag *<c:forEach>*, který bude iterovat napříč kolekcí a pomocí tagu *<c:out>* vytvářet postupně tabulku osob. Díky samotné obecnosti standardu JSTL je možno integrovat složitější komponenty přehlednou cestou a podporovat znovupoužitelnost. Dobrým příkladem je komponenta pro volbu kalendáře:




```
<rich:calendar value="#{calendarBean.selectedDate}"
    locale="#{calendarBean.locale}"
    popup="#{calendarBean.popup}"
    datePattern="#{calendarBean.pattern}"
    showApplyButton="#{calendarBean.showApply}" cellWidth="24px" cellHeight="22px" style="width:200px"/>
```

Obr. 7-17: Kalendář pomocí JSTL

7.7 Enterprise JavaBeans (EJB)

Enterprise JavaBeans byly představeny jako specifikace v roce 1997 společností IBM. Jedná se o serverovou komponentní architekturu pro budování škálovatelných aplikací. Volně lze říci, že jsou to třídy vykonávající aplikační logiku běžící ve speciálním kontejneru na aplikačním serveru. V porovnání se servletovým kontejnerem nabízí EJB kontejner velké množství funkcionalit, které lze využít při tvorbě webové aplikace. Oba kontejnery, jak servletový tak EJB, mohou běžet na stejném aplikačním serveru. Služby EJB kontejnerů jsou definovány ve specifikaci pro danou verzi (aktuální verzi 3.1 se věnuje JSR 318). EJB se běžně používají jako vrstva zajišťující persistenci (zpravidla objektově relační mapování) a pro proces vyžadující transakční zpracování.

Použití EJB v architektuře webové aplikace má dvě hlavní výhody:

- Přenositelnost - pokud již komponentu vytvoříte, můžete ji použít u jiné aplikace na jiném aplikačním serveru.
- Jednoduchost vývoje - EJB kontejner nabízí velké množství služeb, které ulehčují vývoj aplikace. Není vyjímkou, že tvůrci aplikačních serverů přidávají do EJB kontejneru vlastní proprietární rozšířenou funkcionalitu.

7.7.1 Architektura a typy EJB

Verzí EJB je za dobu jejich vývoje přirozeně více, ale v praxi se nejvíce používají verze 2.1 a 3.x. Verze 2.1 definovala tyto základní prvky:

- **EJB** - komponenta s programovým kódem jazyka Java, která vykonává

aplikační logiku (např. počítá čtvrtletní uzávěrku).

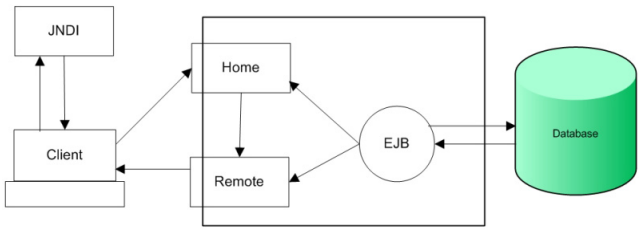
- **Remote Interface** - specifikuje vzdálene metody, které může klient volat.
- **Remote Home Interface** - specifikuje metody, které používají vzdálení klienti pro řízení instance EJB komponent.
- **Deployment descriptor** - XML soubory popisující vlastnosti EJB (např. jakého je typu).

Pro lokální klienty jsou pak definovány další rozhraní:

- **Local Interface** - jedná se o odlehčenou verzí Remote Interface, kterou využívají lokální klienti. Obsahuje metody aplikačni logiky EJB, ke kterým můžou přistupovat lokální uživatelé.
- **Local Home Interface** - obdoba Remote Home Interface ovšem používají ji pouze lokální klienti.

Pro použití EJB s webovými službami byl definován specifikací tzv. Service Endpoint Interface, který nabízí komponenty jako webové služby popsané WSDL dokumentem.

Aby bylo možno komponenty v rámci EJB kontejneru najít, je k dispozici jmenný prostor Java Naming and Directory Interface (JNDI). Celá architektura by pak mohla vypadat například takto:



Obr. 7-18: Architektura EJB (Zdroj: http://sanjaal.com)

Klient v registru JNDI nalezne odkaz na objekt EJB, který chce využít. Poté vytvoří jeho instanci pomocí rozhraní Home. Poté použije rozhraní Remote pro volání metod. Pro komunikaci se využívá již dříve popsaná technologie RMI, konkrétně RMI-IIOP. Rozhraní Home může kromě vytváření instancí samotného objektu také realizovat tzv. pooling, tedy kontrolovat počet vytvářených instancí objektů a tím zrychlit celkový běh aplikace.

Specifikace definuje tři typy Beanů:

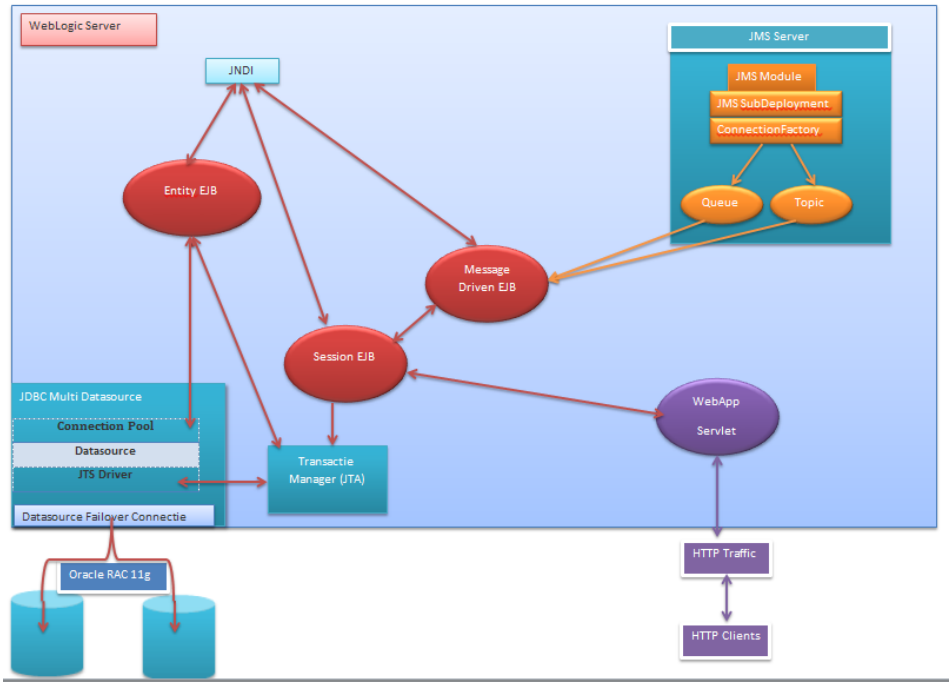
- **Session Bean** - jsou určeny pro ulehčení programátorům vytvořit tzv. middleware (softwarovou vrstvu mezi prezentační logikou a databází) a také ulehčení přístupu ke službám, které poskytuje middleware. Dělí se dále na:
  - Stateless - jsou určeny pro tvorbu komponent, nemají žádný stav, takže nejsou závislé na přihlášeném uživateli ani aktuálním stavu aplikačního serveru.
  - Statefull - uchovávají svůj stav, který si drží po nějaký čas (typicky po dobu přihlášení uživatele).
- **Entity Bean** - používají se pro implementaci persistence do aplikace (pro práci s databází). Na výběr jsou dva druhy - persistence řízená EJB

kontejnerem a persistence řízená samotným Beanem. Ve verzi EJB 3 jsou nahrazeny Java Persistence API.

- Container managed persistency
- Bean managed persistency

- **Message driven Bean** - oproti výše zmíněným typům jsou aktivovány posláním zprávy (nikoli voláním metody). Využívá se zde specifikace Java Message Service (JMS). Navíc neobsahuje Home ani Remote rozhraní. Při inicializaci pouze namapuje na JMS a poslouchá až přijde zpráva, která vyvolá určitou akci.

Po uvedení všech typů EJB je tedy zbývá umístit do kontextu celé architektury vč. aplikačního serveru:



Obr. 7-19: Architektura EJB začleněná do AS Weblogic (Zdroj: http://technology.amis.nl)

Při používání standardu EJB 2.1 si vývojáři často stěžovali na zbytečnou složitost a nepřehlednost. Proto ve verzi 3.0 došlo k zásadnímu zjednodušení celého konceptu EJB. Mezi hlavní zlepšení můžeme uvést těchto pět bodů:

- Deployment deskriptory v XML souborech jsou velice složité a špatně se udržují. V nové verzi jsou metadata, která původně vyjadřovaly XML soubory, přesunuty přímo do tříd a zapsány pomocí anotací.
- Rozhraní Home a Remote zbytečně zesložiťují kód a proto v nové verzi již nejsou vyžadovány.
- Původní filosofie Entity Beanů (CMP, BMP) byla nahrazena Java Persistence API.
- Není již nutně vyžadováno implementovat metody, které často zůstávali prázdné (ejbCreate, ejbPassivate).
- Dříve neexistovala dědičnost mezi EJB komponentami, nová specifikace toto povoluje.

7.8 Nástroje pro urychlení vývoje

Ještě než popíšeme nástroje pro urychlení vývoje, je dobré si nejprve popsat samotný proces, který je potřeba pro vytvoření webové aplikaci a umístění na aplikační server. Každá webová aplikace má předepsanou strukturu nutných adresářu a souborů (např. každá aplikace musí obsahovat adresář /WEB-INF). Abychom mohli aplikaci umístit na server, je potřeba nejprve zkompilovat zdrojové kódy (třídy java, servlety, JSP stránky) a vytvořit soubor s předepsanou strukturou - Web application ARchive (WAR). Protože Java třídy jsou zpravidla v jiných adresářích než JSP stránky, museli bychom postupně procházet všechny adresáře obsahující kód ke kompilaci a nakonec ještě spustit příkaz, který vytvoří WAR archiv. Tento postup bychom museli provádět pokaždé při sebemenší změně aplikace. V nadaci Apache při vývoji jejich webového serveru přišli na to, že pokud si vytvoří nástroj, který automatizuje tyto kroky, urychlí to podstatně vývoj produktu. Proto vytvořili nástroj jménem **Ant**.

Apache Ant

Ant je tedy softwarový nástroj na automatizované sestavování (build) aplikací. Svým způsobem je nástroj podobný nástroji *Make*, který je znám z Unixových systémů. Nástroj je platformě nezávislý a pro sestavení projektu používá uživatelem definovaný XML soubor. Tento soubor obsahuje párové a nepárové tagy pro definici toho, co se má udělat. Většinou existuje pro celý projekt jeden XML soubor a jednotlivé kroky jsou v něm děleny na cíle (target). Proces sestavení se spouští z příkazové řádky s požadovanými parametry.

```
<?xml version="1.0" encoding="UTF-8"?>
...
    <!-- Vytvori WAR archiv -->
    <target name="war" depends="compile,copy-web-xml,compile-jsp">
        <delete file="${cls.dir}/${ant.project.name}.war" quiet="true" />
        <war warfile="${cls.dir}/${ant.project.name}.war"
             webxml="${web.dir}/WEB-INF/web.xml">
            <fileset dir="${web.dir}">
                <exclude name="WEB-INF/**/*" />
                <exclude name="**/*.jxml" />
                <exclude name="**/*.jsp" if="compile.jsp.true"/>
                <exclude name="**/*.inc"  if="compile.jsp.true"/>
                <include name="**/*"/>
            </fileset>
            <classes dir="${cls.dir}">
                <include name="cz/osu/infs2/servlets/*.class" />
                <include name="cz/osu/infs2/taglib/*.class" />
                <include name="cz/osu/infs2/view/*.class" />
                <include name="**/compiledjsp/**/*.class" />
            </classes>
            <webinf dir="${web.dir}/WEB-INF">
                <include name="jboss-web.xml" />
            </webinf>
        </war>
    </target>
...
```

Obr. 7-20: Ant - XML pro sestavení WAR archivu

Výše uvedený obrázek znázorňuje výřez z XML souboru, který sestavuje projekt do WAR archivu. Standardně jsou tyto soubory pojmenovány jako *build.xml* a jsou umístěny v kořenovém adresáři webové aplikace. Cíl se jmenuje "war" a před jeho spuštěním je třeba vykonat cíle "compile, copy-web-xml, compile-jsp". Stačí tedy do příkazové řádky zadat "ant war" a budou automaticky spuštěny všechny předchozí deklarované cíle. Po provedení závislostí je smazán předchozí WAR archiv a sestaven archiv nový. Použité tagy XML dokumentu jsou popsány v dokumentaci. Tímto způsobem lze popsat pomocí tagů a závislostí celý proces sestavení a spouštět ho jediným

příkazem z řádky. Pouze jediným příkazem tak smažete původní aplikaci z aplikačního serveru, zkompilujete znovu zdrojové kódy, sestavíte archiv a poté ho umístíte na aplikační server.

Při vývoji v techologii Java je často potřeba využít externích knihoven a různých frameworků. Každá knihovna znamená návaznost na další projekt. Protože Ant byl orientován pouze na automatizaci jednoduchých procesů, bylo potřeba nalézt nástroj, který zachytí návaznosti projektů a bude procesně orientován.

Apache Maven

Komunitou je nástroj Maven často označován jako "Ant na steroidech". Jedná se tedy o podobnou filosofii - projekt je popsán XML sestavovacím souborem, ovšem v případě Mavenu jsou navíc uvedeny závislosti na projekty jiné. Pokud chci například využít v mém projektu JUnit pro testování, stačí do sestavovacího souboru umístit následující kód:

```
<dependencies>
  ...
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.1</version>
      <scope>test</scope>
    </dependency>
  ...
</dependencies>
```

Obr. 7-21: Maven - závislosti

Maven se postará o stažení potřebných knihoven a integraci JUnitu do projektu. Rozdíl mezi Antem poznáte kromě samotné struktury také v názvu XML souboru - pro Maven se používá soubor *pom.xml* (POM - Project Object Model).

Při budouvání webové aplikace je vhodné začít právě Mavenem, obsahuje totiž šablony pro vytvoření základní struktury webového projektu pod určeným frameworkem. Maven za vás vytvoří předepsanou adresářovou strukturu, stáhne potřebné knihovny daného frameworku a často také vytvoří základní soubory pro prezentaci funkce frameworku - Hello world.

7.9 Nástroje pro podporu testování

Pro standardní unitové testování Java aplikací nám postačí prověřený nástroj JUnit. Automatizované testování webových aplikací je složitější v tom, že výstup je zobrazován ve webovém prohlížeči, který do jisté míry sám ovlivňuje zobrazení samotné stránky. Proto je vhodné prohlížeč do procesu testování zapojit také.

Selenium

Jedna z možností jak funkčně testovat webovou aplikaci je použít Selenium. Jedná se o testovací framework určený přímo pro webové aplikace a podporuje platformy Java, .NET, PHP, Python, Ruby a Perl. Zavedení automatizovaného testování do projektu probíhá v následujících krocích:

1. Tester pomocí pluginu nahraje sled kroků pro testování. V praxi to probíhá tak, že si nainstaluje potřebný plug-in do prohlížeče, spustí záznam a kliká podle scénáře na zobrazenou HTML stránku v prohlížeči.

2. Selenium převede nahrané makro do zdrojového kódu daného jazyka (v našem případě Java).
3. Tester vytvoří testovací třídu a umístí do ní vygenerovaný kód. Pro použití kódu je potřeba importovat Selenium API.

Nyní jsou, podobně jako při použití JUnit, vytvořeny třídy, které lze automatizovaně spouštět jako testové a vyhodnocovat jejich výsledky.

Pro testování se také často využívají integrační servery. V současné době jsou nabízeny řešení označovaná Platform as a Service (PaaS) a jedná se o službu cloud computingu, která uživatelům nabízí prostředí pro deploy jejich aplikací a kontrolu nad procesem deploye vč. nastavení. Jako příklad můžeme uvést Heroku či OpDemand, oboje řešení jsou ovšem komerčního charakteru.

7.10 Java a frameworky

Platforma Java EE je ve své specifikaci vcelku obecná a dobře škálovatelná. Nemůže ovšem sama o sobě v praxi stačit k vytvoření aplikaci v konkurenčně přijatelné cenové hladině. Spoustu úkonů se při vývoji webových aplikací opakuje. Většinou potřebujete vyřešit jako programátoři persistenci aplikace či zpracování požadavků z webového prohlížeče. Tyto problémy se s většinou projektů opakují a tak byly vytvořeny části kódu (frameworky), které tyto problémy řeší a jsou znovupoužitelné v rámci projektů. Proto se v Javě dost často setkáte s modulárním přístupem, kdy jeden projekt je složen z více frameworků svázaných do sebe tvořící webovou aplikaci. Mezi hlavní výhody lze zařadit hlavně:

- Zrychlení vývoje - programátor nevytváří vlastní servlet pro zachytávání požadavků, využije stávající řešení.
- Zavedení ověřených principů - frameworky jsou založeny na dlouhodobé zkušenosti vývojářů a zpravidla obsahují tzv. "best practices" v dané oblasti.

Na druhou stranu použití frameworků může mít za následek tyto problémy:

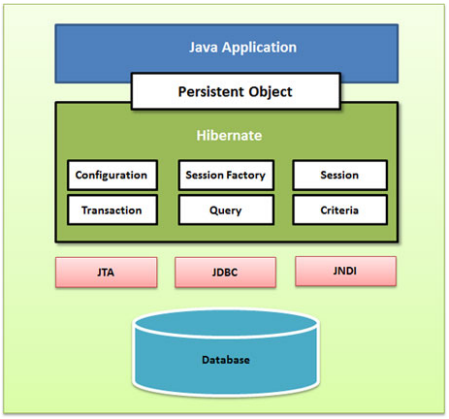
- Obcházení architektury - programátor není dobře seznámen s funkcí frameworku a tak implementuje vlastní řešení, které framework obchází.
- Kompatibilita - jednotlivé frameworky se sebou navzájem nemusí být kompatibilní, jiné vyžadují speciální nastavení. Toto může mít za následek podstatné zdržení při vývoji.

My se zaměříme převážně na framewory pro objektově relační mapování a webové frameworky. Pro velký počet webových frameworků vybereme jen některé, jejich kompletní přehled lze nalézt na adrese:
http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks,
včetně porovnání jejich vlastností.

7.10.1 Hibernate

Hibernate je framework pro objektově relační mapování (ORM) v Javě. Poskytuje funkčnosti, které nám umožní mapovat doménový objektový model do relační databáze (odděluje logický a fyzický datový model). Z našeho kódu tedy můžeme odstranit persistentní vrstvu, která se stará o transformaci a

ukládání objektů do databáze. Hibernate je Open source projekt (GNU Lesser GPL).



Obr. 7-22: Architektura Hibernate (Zdroj: http://www.tutorialspoint.com)

Hibernate je tedy umístěný mezi vaší aplikací a přístupem do databáze. Každý objekt musí obsahovat konfigurační soubor, který mapuje konkrétní datové typy třídy na databázovou tabulku (dříve pomocí XML souborů, nyní pomocí anotací), zároveň také implementuje metody, které persistenci řídí. Pokud na objekt zavolá programátor metodu *update()*, Hibernate vygeneruje SQL dotaz podle dialektu příslušné databáze a objekt sám uloží. Programátor se také nemusí starat o vytváření spojení do databáze a jeho pooling, Hibernate využívá tzv. datových zdrojů (data source), které mají toto na starost.

Pro čtení informací z databáze se využívá Hibernate Query Language (HQL), ze kterého si vzal inspiraci i jazyk Java Persistence Query Language (JPQL). Lze jej popsat jako spojení jazyka SQL a objektové notace se všemy prvky jazyka SQL (vnitřní spojení, vnější spojení, atd.).
Pokud bychom chtěli najít jméno kočky v tabulce všech koček, zapsali bychom to nejspíše takto:

```
select kocky.jmeno from TabulkaKocek kocky
where kocky.jmeno like '%Mikes%'
```

Všimněte si, že zde lze používat tečkovou notaci, výsledek tohoto volání je pak instance objektu se všemi atributy načtenými z tabulky *TabulkaKocek*. Hibernate také odstiňuje programátora od konkrétní implementace SŘBD, aplikace je proto velmi dobře přenositelná napříč různými databázemi.

7.10.2 MyBatis (iBATIS)

Stejně jako Hibernate je MyBatis framework realizující persistenci objektů do relačních databází. Stejně jako Hibernate uchovává mapování v XML souborech nebo anotacích. Podstatný rozdíl je ovšem v tom, že nemapuje objekty na tabulky, ale metody na přikazy SQL. Díky tomu je dovoleno použít výhody konkrétní databáze (procedury, pohledy, PL/SQL funkce a jiné proprietární řešení) a tím podstatně zrychlit chod aplikace. Při mapování pomocí anotací může kód vypadat například takto:

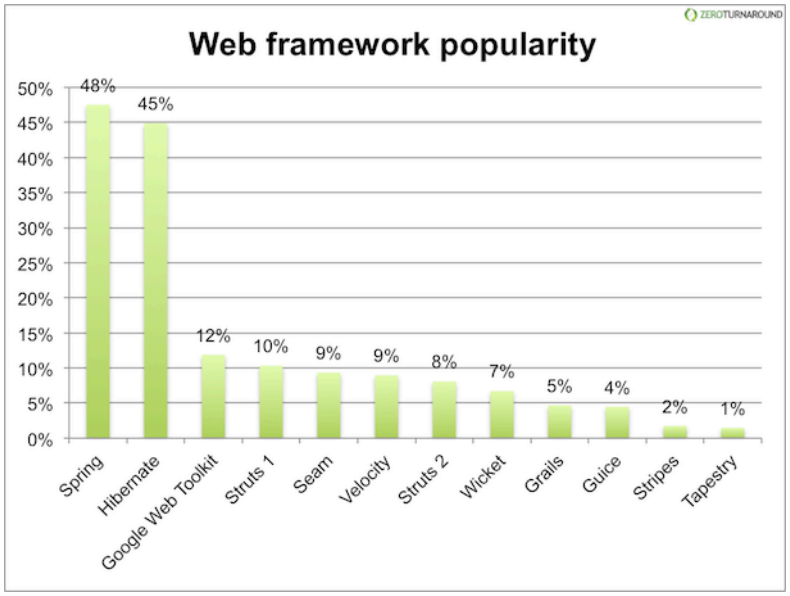
```
public interface rozhraniDAO {
     @Select("Select * from ucitel")
     List<Ucitel> selectAllUcitele();
}
```

Obr. 7-23: Mapování SQL na metodu

Jak Hibernate tak MyBatis vyžaduje pro mapování metadata (použitý datový typ atributu v tabulce, velikost, atd.), které lze přečíst přímo z databáze. Existuje proto spoustu nástrojů, které generují z databáze třídy včetně mapování. Pro složitější datové struktury je nezbytné, aby relační model databáze obsahoval nezbytné návaznosti (cizí klíče, alternativní klíče).

7.10.3 Webové frameworky

Webových frameworků je na trhu celá řada. Hlavní problém vývojáře je rozhodnout se, který framework je vhodný pro tvorbu konkrétního projektu. Pokud bychom se podívali na současná řešení, nalezli bychom třeba tento graf popularity webových frameworků:

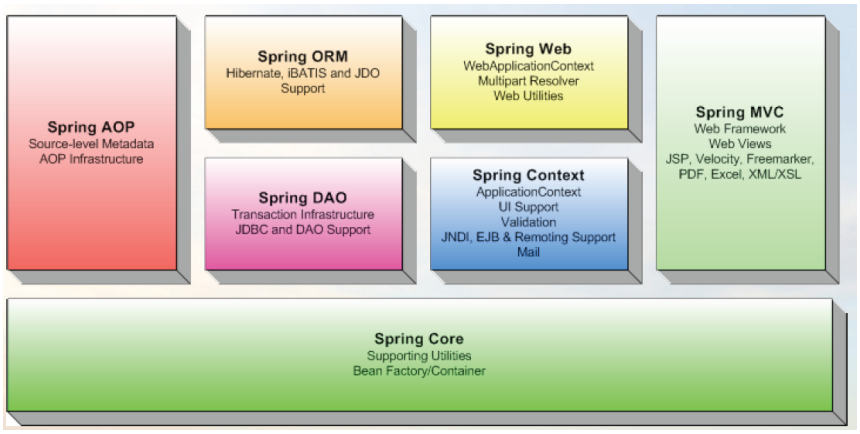


Obr. 7-24: Popularita webových frameworků (Zdroj: http://zeroturnaround.com)

Když pomineme, že je zde prezentován Hibernate jako webový framework, nabyli bychom dojmu, že nejlepší volbou pro náš projekt je Spring, protože ho používá téměř polovina trhu. Na trhu ovšem existuje frameworků více a každý má své přednosti. Obecně tedy při výběru frameworku postupujeme tak, že si nejprve napíšeme požadavky na architekturu (je potřeba persistence?, lokalizace?, bude se používat AJAX?). Poté vyběreme 3-4 vyhovující frameworky a v rozmezí týdne zkusíme vyrobit kostru stejné aplikace ve všech. Framework, ve kterém je vývoj nejsnažší a který nejlépe vyhovuje požadavkům se poté použije jako základ architektury pro novou webovou aplikaci. Ve skriptech se omezíme pouze na ty nejznámější na trhu, všechny frameworky zde zmíněné jsou open source, můžete se tedy podívat na jejich zdrojové kódy.

7.10.4 Spring / Spring MVC

Spring framework je v současnosti nejvíce používaný framework pro tvorbu webových aplikací. Je oblíbený zvláště pro svou modularitu. Dříve se jednalo o projekt Spring MVC sloužící jako podpora návrhového vzoru ve webových aplikacích, tento projekt se později transformoval na jednu ze služeb postavenou nad jádrem Spring. Samotný Spring Framework nepředepisuje přímo žádný programátorský model, zato však nabízí spoustu komponent pro sestavení funkce na míru projektu. Architektura je tedy modulární, zobrazuje ji následující obrázek:



Obr. 7-25: Spring a jeho součásti (Zdroj: http://www.developersbook.com)

Povinná součást každé aplikace je jádro Spring. To obsahuje kontejner podporující techniku Inversion of Control (IoC). Pomocí IoC je kontejner schopen s využitím reflexe spravovat životní cyklus objektů. V praxi to znamená, že pokud programátor potřebuje instanci nějakého objektu, vytvoří tuto instanci za něj samotný kontejner a také spravuje celý životní cyklus této instance. Třída je pak nalezena pouze podle svého jména. Tento způsob práce s objekty je nazýván dependency injection a dokáže zpřehlednit architekturu aplikace, na druhou stranou jí programátor musí dobře ovládat.
Nad jádrem Springu je k dispozici plno služeb, které se integrují podobně jako plug-iny, nejpoužívanější jsou:

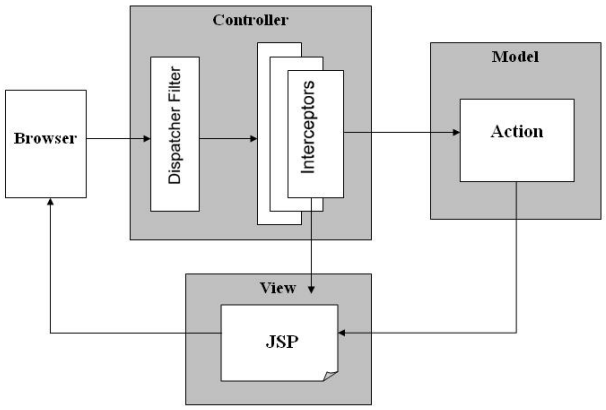
- **Model-view-controller** - framework zajišťující implementaci návrhového vzoru MVC do architektury aplikace, využívá technologii servlet.
- **Data access** - framework pro podporu persistence, podporuje zejména JDBC, Hibernate, MyBatis, JPA.
- **Security** - původně samostatný projekt, podporuje standardy, protokoly a nástroje pro autorizaci a autentizace uživatele.
- **Transaction management** - podporuje transakční zpracování na všech úrovních.

7.10.5 Struts / Struts 2

Dobrou alternativou ke Springu je webový framework Apache Struts. Struts je striktní MVC framework a tak dodržuje přísné rozdělení aplikace na:

- **Model** - reprezentuje datovou část,
- **view** - HTML stránka poslaná uživateli,
- **controller** -akce, které posílají data mezi modelem a view.

Toto rozdělení pak kopíruje i architektura, kterou ukazuje následující obrázek.



Obr. 7-26: Architektura Struts 2 (Zdroj: http://www.javaorigin.com)

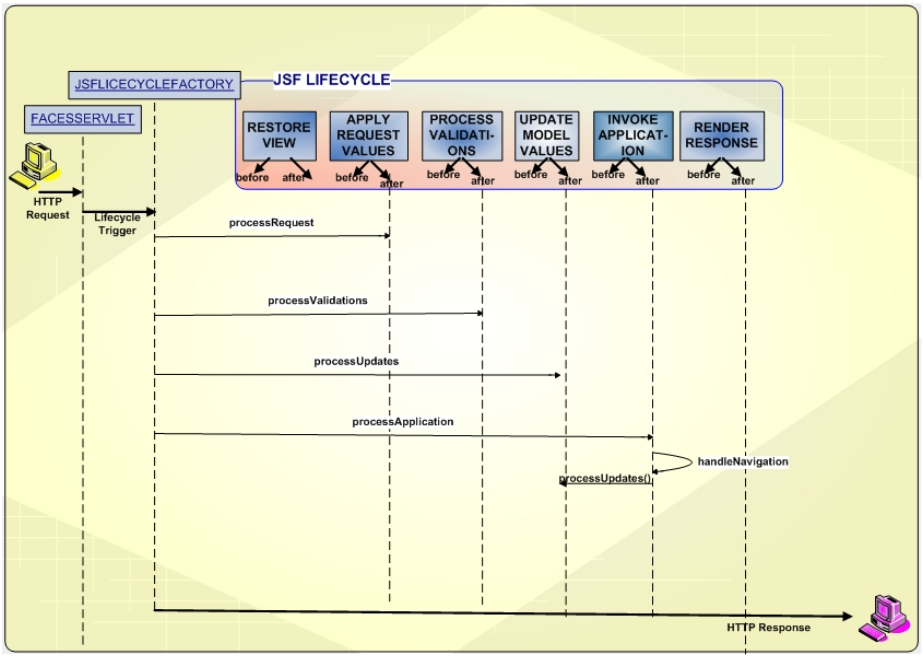
V jednom obsloužení uživatele se využijí postupně 3 návrhové vzory. Uživatel vyšle požadavek na server, který zachytí Dispatcher Filter (první), ten volá podle akce příslušný interceptor (druhý). Interceptor postupně volá registrované akce, protože pouze akce mohou měnit model. Po vyčerpání akcí se řízení předá view a pomocí JSP stránky se vykreslí výsledek uživateli HTML stránkou. Aplikační logika je tak striktně oddělena od modelu a tvoří třetí návrhový vzor - MVC.

V současnosti je aktuální verze 2.3.4, rozdíly mezi verzí 1.x nejsou velmi zásadní. Namátkou se změnil přístup k práci s akcemi. Ve Struts verze 1 jsou všechny akce instanciovány jako jedináček (návrhový vzor Singleton), jedna instance tak obsluhuje všechny volání na danou metodu. Ve Struts 2 se s každým voláním vytváří instance nová a tak odpadá problém se synchronizací vláken.

Struts lze stejně jako Spring rozšiřovat, zde se nejedná o služby nad obecným jádrem, ale o konkrétní plug-iny. Nemění tak architekturu stanovenou frameworkem, pouze dodávají další funkcionalitu.

7.10.6 JavaServer Faces

JavaServer Faces (JSF) je request-driven MVC webový framework zaměřený na jednodušší tvorbu webového uživatelského rozhraní. Předchozí frameworky řeší hlavně architekturu a celkový životní cyklus aplikace, JSF je zaměřen hlavně na prezentační vrstvu. Architektura frameworku je ve své podstatě velmi jednoduchá, skládá se pouze ze servletu nazývaného FacesServlet, aplikační logiky (tzv. backing beans) a stránek zobrazovaných pomocí technologie nazvané facelety (Facelets). Místo architektury je tedy zobrazen životní cyklus frameworku:



Obr. 7-27: Životní cyklus JSF (Zdroj: http://www.javaworld.com)

S frameworkem je programátor schopen vytvořit vcelku rychle fungující webovou aplikaci, pro rozsáhlejší projekty se ovšem tento framework nedoporučuje, protože se aplikace stává s přibývajícími třídami (backing beans) nepřehledná. Často se ovšem JSF kombinuje s jinými frameworky, např. se Springem.

Zajímavé jsou na této technologii především facelety. Při tvorbě webových aplikací je člověk nucen kontrolovat zobrazení výstupu napříč webovými prohlížeči i různými zařízeními (smartphone, tablet). Facelety aktivní ovládací prvky stránky od HTML zobrazení a vytvářejí tak obecné komponenty o jejichž vykreslení (rozumějte konverzi do HTML) se starají samotné facelety. Ty při vykreslení provádějí potřebnou optimalizaci zobrazení pro konkrétní zařízení. Standardně měly zobrazovací soubory s facelety příponu .jsf či .xhtml, v současnosti se používají i v souborech JSP.

Aby bylo možno facelety v projektu použít, je potřeba namapovat příslušný servlet v souboru web.xml webové aplikace (využijeme obecnosti technologii servlet zmíněné výše):

```
<servlet-mapping>
     <servlet-name>Faces Servlet</servlet-name>
     <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

Poté již v XHTML souboru definujeme zobrazení v souladu se specifikací. Pro vytvořeni pohledu využíváme tagy z příslušného API, které jsme importovali na začátku souboru:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t"%>
```

```
<h:form>
    <h:outputText value="Vítejte uživateli #{loggedInUser.name}" disabled="#{empty loggedInUser}" />
    <h:inputText value="#{bean.property}" />
    <h:commandButton value="OK" action="#{bean.doSomething}" />
</h:form>
```

Výše uvedené řádky kódu vykreslí HTML dokument uživateli. Tag <h:outputText> vypíše pozdrav, pokud je uživatel přihlášen. Zde je umístěna jistá logika - tag kontroluje přihlášení uživatele. Nejedná se ale o aplikační logiku, ale pouze o logiku zobrazovací. View tak zůstává důsledně odděleno od aplikační logiky aplikace. Tag <h:commandButton> odesílá formulář s akcí, která iniciuje volání metody *doSomething* ve třídě nazvané *bean*.

Po uvedení konceptu faceletů vývojáři začali nabízet své vlastní zobrazovací komponenty, nejznámější jsou Apache Tomahawk a IceFaces. Druhé jmenované si můžete vyzkoušet na adrese http://icefaces-showcase.icesoft.org/.

Kontrolní otázky:

1. Jaké edice Javy znáte a k čemu slouží?
2. Jaké jsou hlavní technologie Java EE?
3. Jaké webové frameworky pro podporu tvorby aplikací v Javě znáte a jak se mezi sebou liší?

Úkoly k zamyšlení:

V poslední části o Javě jsme ukázali příklady některých frameworků. Zamyslete se nad tím, v čem je podle vás obrovské množství podpůrných nástrojů tohoto typu dobré a v čem špatné. Srovnejte situaci v produkty Microsoftu pro podporu vývoje v aplikací v .NET. Co vyhovuje více Vám? Možnost volby nebo jasně dané prostředí bez možnosti změny?

Korespondenční úkol:

Pokuste se navrhnout blokovou architekturu budoucí aplikace s využitím představených Java technologií a popsaných frameworků. Existují následující požadavky: bude se jednat o elektronické bankovnictví přístupné jak z běžného PC, tak z mobilního telefonu. Na serverové straně musí být škálovatelné a rozšiřitelné a musí být možné je provozovat na různých OS.

Shrnutí obsahu kapitoly

Kapitola se zabývala popisem edicí Javy s důrazem na platformu Java Enterprise Edition a jejími specifikacemi. Představili jsme JSP, servlety a EJB. Dále jsme stručně popsali možnosti nástrojů pro automatizované sestavení a testování webové aplikace postavené na platformě Java.
Poslední část se zabývala popisem důležitých frameworků pro tvorbu aplikací v Javě.

8 Webové služby, SOA

V této kapitole se dozvíte:

- Co jsou webové služby a k čemu je lze použít
- Jak získat, zjistit definici a komunikovat s webovou službou
- Jak tvořit webové služby, co je RPC, SOA, REST
- Jak se vytváří a používají aplikace na architektuře SOA
- Co je BPEL a jeho vztah k SOA

Po jejím prostudování byste měli být schopni:

- Chápat principy práce webových služeb
- Znát výhody a nevýhody jednotlivých způsobů implementace webových služeb
- Vysvětlit význam pojmů WS, RPC, SOA, REST, SOAP, WSDL, UDDI, WSIL

Klíčová slova této kapitoly:

SOA, WS, Webové služby, RPC, REST, SOAP, WSDL, UDDI, WSIL

Doba potřebná ke studiu: 4 hodiny

Průvodce studiem

Kapitola představuje problematiku webových služeb, jejich návrhů, principů komunikace a implementace. Dále vysvětluje termín SOA, principy a způsoby práce s aplikacemi na této architektuře a rozšíření BPEL.
Na studium této části si vyhraďte 4 hodiny.

8.1 Webové služby

Webové služby jsou moderní technologií skloňovanou v mnoha oblastech vývoje informačních systémů v poslední době. O co vlastně jde?
Webová služba je softwarový systém určený k podpoře interoperací dvou počítačů přes počítačovou síť. Jedná se vlastně o webové aplikační rozhranní, kte kterému ostatní počítače přistupují pomocí počítačové sítě (zpravidla Internetu). Implementace je nejčastěji pomocí dvouvrstvé architektury klient – server. Počítač – server – nabízí určitou naprogramovanou funkčnost ve formě metody. Tato metoda není přístupná pouze z daného počítače, ale pomocí technologie webových služeb ji lze zavolat z jiného počítače v síti. Počítač – klient musí zjistit, kde se daná metoda nachází a jak ji zavolat – následně danou metodu spustí a od serveru obdrží výsledky dané funkčnosti.
Příklad: Firma X nabízí na svém serveru webovou službu pro zjištění zboží na skladě. Webová služba (jelikož je to z pohledu programování metoda) vyžaduje na vstupu parametr *kód_zboží* a vrací zpět celé číslo označující počet položek na skladě. Dodavatel zboží firmě X chce zjistit, zda již je třeba dodat nové zboží. Zavolá tedy danou webovou službu a získá počet zboží na skladě.
Implementace této funkčnosti je samozřejmě řešitelná různými způsoby, ne nutně webovými službami (např. speciální HTML stránka), nicméně webové služby nabízejí díky způsobu své realizace určité vlastnosti, které ostatní řešení

nabízet nemohou (např. HTML stránka nevrací meta-informace o svém obsahu). Jako základní výhody lze zmínit:

- Multiplatformnost – při propojení klienta s poskytujícím serverem typicky nastane případ, když se jedná o propojení různých platforem. Každá platforma běžně nabízí technologie pro nízkoúrovňovou síťovou komunikaci přes síť, ale klasická řešení požadují, aby oba komunikující stroje měli stejnou platformu (například Java – RMI, .NET - .NET Remoting). Webová služba je obecný předpis využívající vlastností současného protokolu http a jazyka XML a díky tomu lze propojit i technologie, které se liší.
- Standardizace – celá problematika webových služeb je standardizována a má jasně definované vstupy, výstupy, způsoby volání a odpovědí. Při korektní implementaci se tak nemusí řešit problémy s kompatibilitou.
- Podpora prostředí – moderní vývojová prostředí a programovací nástroje poskytují podporu pro pohodlný návrh a implementaci serverových i klientských částí webových služeb. Díky tomu většina tvořených implementací vzniká za pomoci průvodců, což snižuje čas a chybovost vytvářených řešení.

Postup zavolání webové služby klientským počítačem lze rozdělit do několika fází (vlastní komunikace dále do způsobů užití):

1. Nalezení webové služby – na kterém počítači a pod jakým názvem se nachází požadovaná funkčnost;
2. Zjištění definice webové služby – jakým způsobem je služba definovaná, jakou má signaturu (tj. jaké má vstupní parametry a jaké hodnoty vrací) a jaké používá ke komunikaci protokoly;
3. Vlastní komunikace – daným protokolem/způsobem:
    a. RPC – Remote Procedure Calls;
    b. SOA – Service Oriented Architecture;
    c. REST – Representational state transfer;
    d. další.

Základy webových služeb se pro lepší přehlednost a případné rozšíření funkcionality sjednocují to tzv. profilů. Cílem je vylepšení vzájemné interoperability a poskytování rozšířených schopností. Profil webové služby se označuje prefixem (WS-* kde * zastupuje identifikátor profilu, např. WS-I, WS-Security, WS-Addresing, WS-BPEL) a definuje mimo jiné specifikaci definice a protokolů komunikace webové služby (WSDL + verze, SOAP + verze). Pro vybrané profily jsou definované případy užití (ulehčující výběr profilu k implementaci) a také testovací nástroje pro ověření funkčnosti profilu. Tato problematika nebude dále blíže představena.

8.1.1 Nalezení webové služby

Potenciální konzument webové služby může získat odkaz k webové službě základními způsoby:

- V globální rejstříku se najde vhodná služba a k ní odpovídající zprostředkovatel – UDDI;
- Nalezne se vhodný zprostředkovatel a u něj webová sluba – WSIL.
- Osobní výměnnou – klient ví, od jakého poskytovatele chce webovou službu získat (poskytovatel poskytne klientovi webovou adresu, na

které se webová služba nachází), případně nalezne její URL adresu na webových stránkách poskytovatele.

První dvě řešení nicméně nejsou standardy W3C, jedná se tedy pouze návrhy řešení a existuje k nim více dalších alternativ (např. DISCO od Microsoftu, které se používá v .NET). V praxi se však nejčastěji v současné době používá třetí způsob.

UDDI – Universal Discovery, Description and Integration

O existující webové službě se klient může dozvědět buď pomocí svého interního zdroje (e-mailem od IT zaměstnance serveru, apod…) nebo ji lze nalézt v rejstříku webových služeb. Tento rejstřík (v XML formátu) má uložený jednotlivé firmy a jejich webové služby v kategoriích pro jednodušší vyhledávání. Samotný rejstřík je fyzicky provozován na několika počítačích – uzlech – které si mezi sebou navzájem sdílejí a aktualizují data.
Tento rejstřík se nazývá UDDI – Universal Discrovery, Description and Integration a vznikl z iniciativy bussines organizací (např. MS, SUN, IBM, HP a další) právě pro publikování jejich webových služeb a způsobu jejich interakce přes Internet. Sestává se ze tří základních komponent:

- Bílé stránky – adresy, kontakty a známé identifikátory;
- Žluté stránky – zařazení firem na základě běžných taxonomií;
- Zelené stránky – technické informace o službách nabízených organizacemi.

V praxi se bohužel funkčnost příliš neosvědčila (se vzrůstajícím počtem služeb se snižovala aktuálnost záznamů). Je to způsobeno hlavně nemožností ověřit důvěryhodnost poskytovatelů služeb.

WSIL – Web Service Inspection Language

WSIL reprezentuje opačný směr vyhledávání webové služby. Nejdříve klient nalezne poskytovatele služby (který je pro něj dostatečně důvěryhodný) a zašle mu požadavek na popis rozhranní. Obecně je popis služeb poskytovatele zpravidla umístěn v souboru *inspection.wsil* v hlavním adresáři web-serveru poskytovatele. Díky tomu na něj dosáhnou prohledávácí a indexovací stroje.

8.1.2 Definice webové služby – WSDL

Jakmile klient ví, kterou webovou službu chce používat, musí zjistit, jakým způsobem ji může zavolat. K určení signatury a dalších (např. meta-) informací o webové služby se používá jazyk WSDL – Web Service Description Language. Je založen na XML (navíc využivá standardy XML Namespaces a XML Schema) a vznikl sloučením jazyků firem IBM (NASSL), MS (SCL) a Ariba (SDL). Pod W3C existuje pracovní skupina, která se stará o vývoj WSDL, nyní ve verzi 2.0 (původně vyvíjen jako 1.2).
Na nejvyšší úrovni jazyk WSDL popisuje jednotlivé služby. U každé služby je dán způsob (protokol) volání (např. SOAP přes HTTP, SOAP přes HTTP nad SSL, …) a přístupová adresa. Dále WSDL obsahuje definici signatury parametrů webové služby a definici návratové hodnoty. Jednotlivé definice

jsou buď jednoduché předdefinované datové typy, nebo typy složené z typů jednoduchých. Samotné WSDL nabízí základní jednoduché typy jako int (a další běžné celočíselné typy), decimal, float, boolean, string, base64Binary (binární data), dateTime (čas), duration (interval) a jejich odvozeniny (výčtové typy, číselné intervaly, záporná čísla). Dokonce lze vyjádřit i dědičnost komplexních typů.
Hlavní výhodou WSDL je, že je to standardizovaný formální jazyk, je tedy pevně daný a formálně zpracovatelný. Výsledkem je, že je možno automatizovaným způsobem vystavět proxy-objekt nad voláním webové služby a z prostředí programu používat tento objekt. Programátorovi tak odpadá celý komplexní balík funkčností zajišťující komunikaci s webovou službou. Prostředí za programátora umí typicky:

- U serveru části vytvořit a publikovat na webovém serveru určitou metodu jako část webové služby a zpracovávat její požadavky. V souvislosti s tím umí i přijímat a odesílat požadovaná data s ohledem na omezení datových typů. Podporuje-li to webová služba, umí navržené řešení i realizovat problematiku ověření a správy stavů.
- U klienta vytvořit základní datové typy (třídy, struktury, …) požadované jako parametry webové služby či jako její návratový typ. Programátor tak nemusí tyto typy psát ručně, ani se starat o mapování na předdefinované datové typy.
- U klienta umí navíc prostředí vytvořit objekt, který umí zabalit, zprostředkovat a vrátit výsledek volání webové služby; a to typicky jak synchronně, tak asynchronně.

8.1.3 Protokoly webové služby, SOAP

SOAP – Simple Object Access Protocol – je protokolem pro posílání objektů / zpráv ve formátu XML. Vznikl jako základní protokol pro vzdálenou komunikaci a zasílání objektů – prvotně použit v řešení vzdáleného volání procedur. Jedna aplikace pošle v XML-SOAP zprávě požadavek aplikaci druhé, ta jej obslouží a vrací výsledek zpět opět ve formátu XML-SOAP. Zpráva je na principu peer-to-peer a protokol uvažuje pouze jednosměrný přenos od odesílatele k příjemci.
První verze vznikla na konci roku 1999 po spolupráci firem DevelopMentor, Microsoft a UserLand jako protokol pro RPC založený na XML a nahrazovala dříve používaný protokol XML-RPC. Verze SOAP 1.1 byla zaslána a akceptována konsorciem W3C.
Fyzicky je zpráva v SOAP jednoduchým XML dokumentem podobnému hierarchii HTML. Nejvyšší element se jmenuje *Envelope* a obsahuje dva vnořené elementy *Header* a *Body*. Nepovinná hlavička obsahuje rozšiřující informace o zprávě (identifikace uživatele, autentizace, …), tělo obsahuje informace přenášené zprávou. Jednotlivé použité tagy jsou adresovány pomocí jmenných prostorů XML. SOAP protokol obsahuje také informaci o způsobu serializace (uložení) dat ve zprávě. Lze posílat jak běžné datové typy (číslo, řetězec, datum, …) tak tvořit komplexní datové typy (podobně jako ve WSDL).
Pro přenost se nejčastěji používá protokol HTTP. Hlavním důvodem je široká podpora HTTP a také možnost správy webové služby libovolným webovým serverem. Možnost komunikace přes http také ulehčuje konfiguraci sítě pro

použití webových služeb. Jednotlivé implementace však samozřejmě podporují i další přenosové mechanismy (např. SMTP, JMS).

RPC – Remote Procedure Calls

Volání vzdálených procedur je první možnost, jak využít webové služby. Jedná se (zjednodušeně) o volání funkce (metody = procedury) na jiném adresovaném místě (typicky další počítač ve sdílené síti). Cílem je, aby programátor mohl požadovanou funkčnost volat stejně jako by ji volal při lokálním použití. Pokud je funkčnost psána pomocí objektově orientovaných principů použití, nazývá se též vzdálené volání metod (Remote Method Invocation).

SOA – Service Oriented Architecture

Jedná se o vyšší úroveň použití RPC. Přístupu SOA je věnována kapitola 5.2.

REST - Representational state transfer

REST je další běžný způsob použití webových služeb. Obecně lze říci, že definuje formát, jakým se webové služby mají pojmenovávat, definovat a chovat, aby sjednotily a ulehčily přístup klientům. Cílem je tedy zajistit "vyšší standard" v pojmenovávání a tvorbě webových služeb. Základním paradigmatem RESTu je tvorba webových služeb orientovaných na operace a stavy objektů (state) a tedy omezit rozhranní webových služeb na pevně dané operace. U objektově orientovaných přístupů se zpravidla vychází z paradigmatu CRUD(L) – cílem je zajistit základní operace s objektem (resp. se stavy objektu):

- C – Create – tvorba nového objektu;
- R – Read – čtení / získání jednoho objektu;
- U – Update – změna objektu;
- D – Delete – smazání / odstranění objektu;
- L – List – čtení / získání skupiny objektů (zpravidla podle určitých kritérií, jedná se o výkonnostní vylepšení pro *Read*).

Není samozřejmě třeba operace realizovat všechny. Server může omezit podporované operace. Výsledkem takového řešení – tzv. *REST-ful* aplikace – je (v případě použití webových služeb) sada webových služeb daných operacemi s objektem a názvem objektu (nebo jiná konvence splňující tuto funkčnost), například:

- CreateXXX (…);
- ReadXXX (int id);
- UpdateXXX (…);
- DeleteXXX (…);
- ListXXX[ByYYY](…);

Obecně se REST aplikace řídí základními principy:

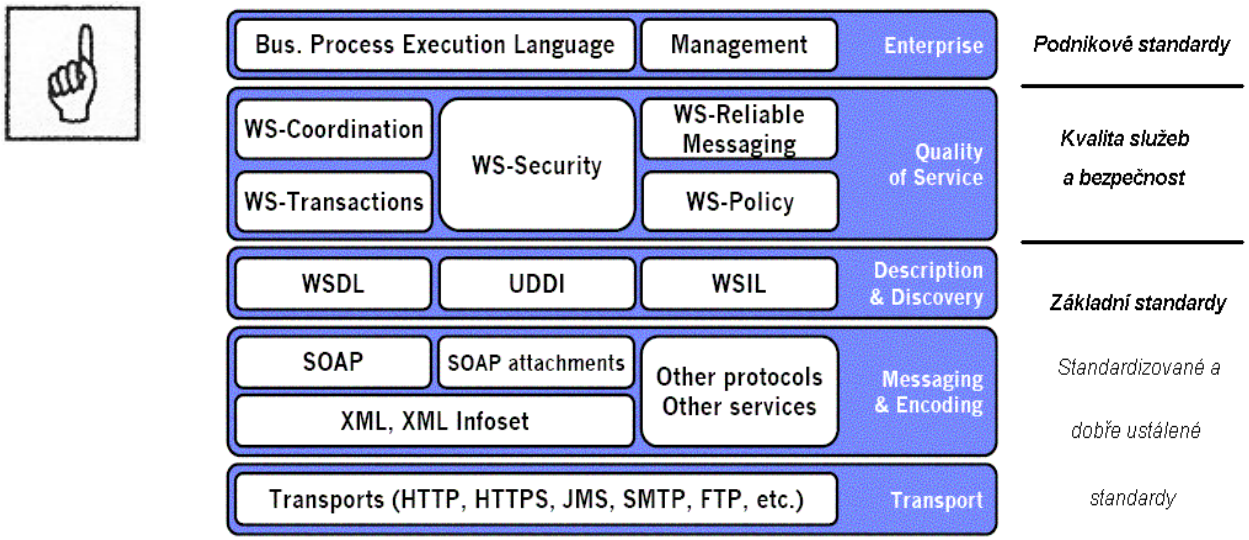
- Stavy a funkcionalita rozdělena do zdrojů
- Každý zdroj jednoznačně adresovatelný přes URI
    - Existuje jednoznačné rozhranní pro přenos dat
    - Existuje množina korektních operací, které lze provést

- Existuje množina typů obsahu (+ code on demand)
- Protokol je typu klient/server, bezstavový, cachovatelný, bezestavový, vrstvený.

Příkladem REST-ful aplikace je například WWW. Stavy a funkcionalita jsou rozděleny do zdrojů – server se svým obsahem – jednoznačně adresovatelné (přes URL/URI) s jednoznačným rozhranním (dle URL HTTP/HTTPS/SMTP) a operacemi k provedení (GET/POST/PUT/DELETE). Množina typů obsahu je zastoupena MIME-typy, příkladem code-on-demand je například JavaScript (dynamický kód stránky lze načíst až na vyžádání – code-on-demand). Protokol HTTP je typu klient/server, bezstatový, vrstvený, výsledek je cachovatelný.

8.2 SOA

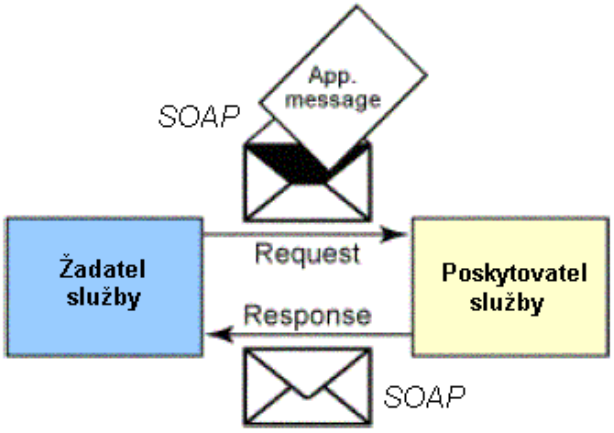
Specifikace webových služeb (WS – Web Services) definuje detaily potřebné pro implementaci služeb a interakci mezi nimi. Servisně orientovaná architektura (SOA – Service Oriented Architecture) je přístup, který slouží ke konstrukci distribuovaných systémů, které poskytují koncovým uživatelům funkci aplikace ve formě služby nebo jsou sami součástí dalších služeb. SOA může být založena na webových službách, ale může také používat jiné technologie. Klíčová je podstata volného spojení služeb (tzv. loose coupling), např. rozhraní služby je nezávislé na implementaci. Vývojáři mohou vystavět aplikace skládáním několika různých služeb bez znalosti jejich implementace. Služba může být implementována technologiemi jako .NET nebo J2EE a aplikace, která službu používá, může být založena na úplně jiné platformě či jazyku. Principem a hlavním přínosem SOA jsou služby a jejich opakovaná použitelnost, která je možná voláním z definice procesu popsaného pomocí jazyka BPEL. Existující standardy používané v SOA architektuře popisuje následující Obr. 8-1.



Obr. 8-1: Standardy SOA

SOA má následující klíčové charakteristiky:

- Služby obsahují platformě nezávislé popisné rozhraní ve formě XML dokumentu, standard pro popis služeb je WSDL – Web Service Description Language.
- Služby komunikují pomocí zpráv definovaných pomocí XML Schema (SOAP), na tyto zprávy může být nahlíženo jako na klíčové zpracovávané podnikové dokumenty.
- Služby jsou v podniku registrovány pomocí registru služeb, aplikace v tomto registru vyhledají danou službu a poté ji zavolají, jako registr služeb se využívá UDDI – Universal Description, Discovery, and Integration.
- Každá služba má definovanou kvalitu služeb (QoS), klíčovými elementy jsou bezpečnostní požadavky – autentifikace a autorizace, spolehlivé doručování zpráv apod.



Obr. 8-2: Příklad komunikace mezi službami

Služba je v servisně orientované architektuře aplikační funkce ve formě znovupoužitelné komponenty pro použití v byznys procesu. Služba také poskytuje informace a usnadňuje změnu dat z jednoho platného, konzistentního stavu do druhého.

Jak již bylo řečeno lze volání či provádění služeb automatizovat pomocí procesního modelu (viz BPEL standard, Obr. 8-1). To je důvod, proč SOA zahrnujeme do procesních přístupů. Vztah mezi webovými službami a jazykem BPEL si stručně popíšeme nyní.
Základními stavebními kameny SOA jsou tedy:

- Aplikace – tvoří nejnižší úroveň architektury, jedná se o ucelené kousky kódu, které vykonávají nějaké funkce.
- Webové služby – tvoří prostřední vrstvu, jedná se o WSDL infrastrukturu (popisy rozhraní), která umožňuje přístup k aplikacím.
- Byznys (obchodní) proces – je nejvyšší vrstvou, proces je definován jako určitý počet kroků, kdy většina z nich je reprezentována jako volání webové služby.

Je zřejmé, že SOA architektura není úplně vhodná pro použití na běžné vnitropodnikové homogenní aplikace. Vhodné je použití SOA pro správu a konstrukci složitých a komplexních distribuovaných aplikací zvláště v heterogenním prostředí, pro dobře strukturované a popsané procesy a jejich

automatizaci (pomocí WS), pro heterogenní procesy, které se často mění (např. finančnictví či telekomunikace) nebo pro systémy, u kterých je vyžadováno skrýt svou implementaci.

Problémem může být implementace některých mechanismů, jako jsou například transakce či dostupnost. Webové služby definují standardy pro atomické (WS-Atomic Transaction) a dlouho trvající obchodní transakce (WS-Business Activity), nemusí však být vždy jednoduché mapovat tyto standardy do jednotlivých transakčních mechanismů databází. SOA také neřeší problémy s dostupností, naopak její nasazení ji může často i zhoršit. Problematika verzí musí být velmi dobře promyšlena ve fázi návrhu, jednou popsané rozhraní služby již totiž nelze měnit, až do doby formálního ukončení služby (depricate).

Kontrolní otázky:

1. Co je to webová služba?
2. Jaký je rozdíl mezi UDDI, WSDL a SOAP?
3. Jaké jsou základní způsoby užití webových služeb?
4. Co je SOA?

Úkoly k zamyšlení:
Pokuste se zamyslet nad cestou, které vedla k webovým službám, jaké byly asi předchozí kroky a jak se budou vyvíjet architektury v budoucnu?

Korespondenční úkol:
Pokuste se zamyslet nad rozdílem v architektuře, údržbě, snadnosti/složitosti návrhu, pokud použijeme k návrhu koncept webových služeb a v případě, pokud budeme vyvíjet aplikaci pomocí klasických technologií jako například C++ a relační databáze. Jaké jsou podle Vás rozdíly ve zmíněných oblastech?

Shrnutí obsahu kapitoly
V této kapitole byl vysvětlen pojem webové služby, představeny základní způsoby získání, definice a komunikace webových služeb. Byly zavedeny pojmy RPC, REST a SOA a vysvětleno jejich použití.

9 AJAX

V této kapitole se dozvíte:

- Jaké jsou vývojové tendence webových aplikací
- Co je to RIA
- Co je to AJAX a jeho použití

Po jejím prostudování byste měli být schopni:

- Znát principy technologie AJAX
- Ukázat příklady použití technologie AJAX

Klíčová slova této kapitoly:

RIA, AJAX

Doba potřebná ke studiu: 2 hodiny

Průvodce studiem

Kapitola představuje problematiku tvorby moderních webových aplikací. Ukazuje na tendence tvorby aplikací pomocí RIA a způsoby a výhody tvorby aplikací pomocí technologie AJAX.
Na studium této části si vyhraďte 2 hodiny.

9.1 Popis technologie AJAX

9.1.1 Rich Internet Application

Rich Internet Application, zkráceně RIA, je směr, kterým se vydává další generace aplikací běžících na standardech internetu. Typické možnosti webových aplikací jsou z hlediska uživatelského komfortu a funkcí limitovány omezeným rozsahem běžně rozšířených technologií internetu. Bylo tedy nutno hledat nové cesty, jak tato omezení odstranit, přičemž Rich Internet Application poskytují nový prostor pro vývoj potřebných řešení.
Stále se zvyšující nároky budou určovat trend vývoje internetových aplikací, a to především v následujících oblastech:

- komplexnost grafického rozhraní (MDI koncept)
- uživatelsky přívětivého chování (odstínění od modelu žádost/odpověď)
- komfort funkcí odpovídající klasickému desktopovému řešení (drag&drop, klávesové zkratky, kontextová nápověda)

Současné aplikace musí řešit dva primární problémy. Za prvé, aplikace se musí srovnat s HTTP protokolem, jeho bezstavovostí a jeho modelem žádost/odpověď. Sebemenší změna stavu (autokompletace dat, validace, aktualizace části dat) na klientu musí vyvolat požadavek na server, který jej musí obsloužit a zpět vrátit všechna předešlá data. Druhým problémem jsou poměrně omezené možnosti prezentačních technologií (HTML, CSS) pro kompaktnější grafická rozhraní.

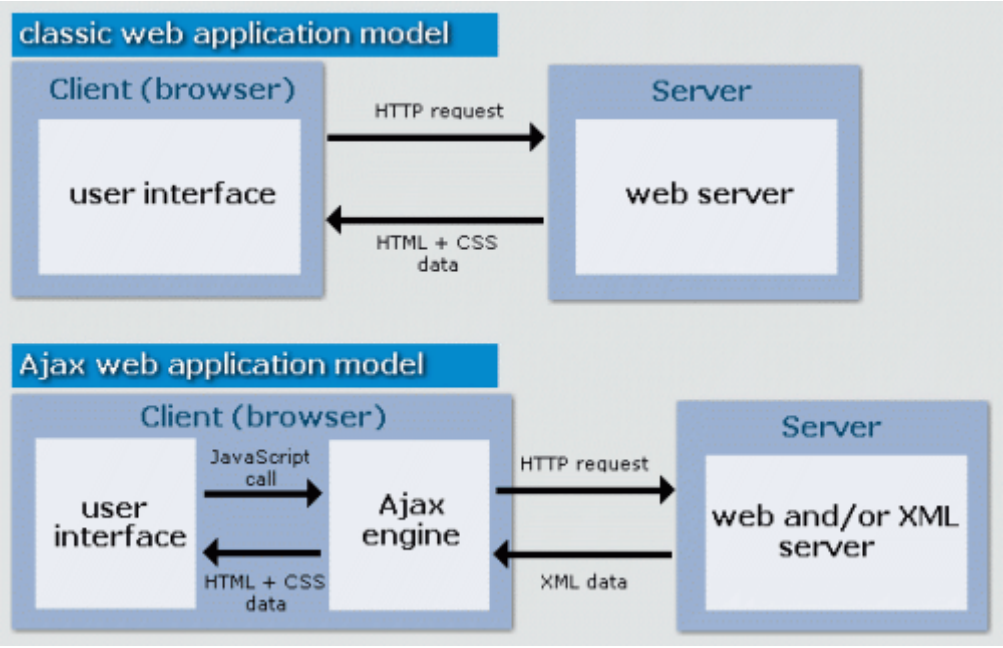
Na jedné straně tak máme určitou množinu požadavků a na straně druhé máme konečné možnosti dnešních technologií. Za Rich Internet Application můžeme považovat takové aplikace, které dokáží splnit ty nejnáročnější požadavky z výše zmíněných oblastí.

9.1.2 AJAX – Asynchronous Javascript and XML

AJAX není sám o sobě implementací technologie či softwarovým produktem, ale jedná se o obecný koncept nebo lépe návrhový vzor pro RIA.
AJAX popisuje obecně způsob tvorby webových aplikací, kdy aplikace prezentuje data pomocí HTML/DHTML a místo načítání nové stránky v reakci na jakoukoliv akci uživatele využívá JavaScriptu a XML k načtení dat bez postbacku celé stránky na server.

Tyto technologie lze použít například k dynamickému načítání různých hodnot ve formuláři, validaci zadaných dat (např. zda již existuje zadané uživatelské jméno), ale i pro tvorbu celých webových aplikací založených na této technologii.

Základním stavebním kamenem je objekt XMLHttpRequest, který umožňuje asynchronní volání serveru. V klasickém webovém modelu každá změna stavu na klientu vyžaduje obnovení celého uživatelského rozhraní. Nejdříve je tedy žádost o změnu stavu, následuje odeslání požadavku na server, vyřízení požadavku a vše končí zasláním kompletního uživatelského rozhraní s daty, přičemž jednotlivé kroky jsou vzájemně synchronizovány. Naopak AJAX, díky XMLHttpRequest, může vyvolat libovolný počet nezávislých požadavků, jejichž výsledky mohou ovlivnit pouze patřičné části uživatelského rozhraní, bez nutnosti jeho celkového znovunačítání. Tedy, žádost o změnu stavu, vygenerování požadavku přes XMLHttpRequest, vyřízení požadavku serverem a zpracování vrácené odpovědi XMLHttpRequestem a změna patřičné části uživatelského rozhraní.



Požadavky z uživatelského rozhraní jsou převedeny na volání metod JavaScriptu v AJAX enginu, který komunikuje se serverem. Server odpovídá

ve formátu XML dokumentu, který AJAX engine transformuje na prohlížečem zobrazitelné HTML a CSS.

9.1.3 Ukázka použití

V dnešní době lze na AJAX narazit na většině moderních webových stránek. Kdekoliv se stránka je schopna dynamicky změnit, aniž by se provádělo zdlouhavé odesílání celé stránky na server a zpět, jedná se o použití technologie AJAX. Typickým příkladem použití konceptu AJAX je např. Google Suggest (http://www.google.com/webhp?complete=1&hl=en). Google stránka vypadá stejně jako obvykle. Pokud se však začne na klávesnici vyťukávat heslo, které je hledáno, prohlížeč napovídá. Nevyužívá se ale historie dotazů z počítače, nýbrž napovídá Google, a to tak, že po jednotlivých písmenkách na pozadí zavolá **webovou službu**[28] na Google serveru, která mu vrátí sadu nejčastěji hledaných hesel podle písmen, která jsou zadána. Podstatné je, že to funguje - a funguje to rychle. A výsledný dojem pro uživatele: stránka se ani nehne, tzn. nenačítá se znovu celá, pouze se obnovují rozbalovací seznamy s nabízenými hesly.
AJAX je moderní technologií a využívá se v množství dalších webových aplikací, například Google Maps – mapy celého světa, portál mapy.cz, Gmail – webový poštovní klient, Kiko – online kalendář, Meebo – instant Messenger, Basecamp – online služba pro správu projektu, Writely – online textový editor, Time tracker – nástroj pro osobní time management a další.
V poslední době umožňuje kombinace rychlých prohlížečů a počítačů, JavaScriptu a AJAXu již vytvářet aplikace, které se velmi přibližují běžným, desktopových aplikacím, ať v rychlosti, nebo v nabízených funkcích.

9.1.4 Technologie

Za AJAXem jsou skryty tyto technologie:

- HTML/XHTML – zažitá prezentace dat na Internetu;
- CSS – stylistické formátování obsahu HTML/XHTML dokumentu;
- DOM – dokument object model – jedná se o model, který celou HTML/XML stránku zpracovává pomocí objektové reprezentace. Skriptovací jazyky (typicky JavaScript) potom může s jednotlivými objekty manipulovat a přizpůsobovat jejich chování, vzhled nebo obsah;
- XML –formát výměny dat;
- XSLT – jazyk transformace XML dokumentů do XHTML dokumentů (obvykle za použití CSS), slouží k jednoduchému přizpůsobení vzhledu stránky na základě daných pravidel;
- XMLHttp – primární komunikační broker;
- JavaScript – skriptovací jazyk pro programování s AJAX engine.

[28] Propojení, kdy AJAXový požadavek volá určitou webovou službu a výslednou stránku zobrazuje podle jejího výsledku, je velmi časté.

Nezbytně nutné jsou však pouze tři technologie: HTML/XHTML, DOM, JavaScript. Na straně serveru zpracovává AJAX požadavky obvykle PHP, Java servlet, JSP, ASP.NET. Další, v tomto ohledu důležitou vlastností AJAXu, je **zajistit plnou podporu vykonávaných operací i v případě, že technologie nejsou dostupné** – tj. typicky, pokud má uživatel starý prohlížeč, případně má vypnutý JavaScript. Tehdy se původně XHTTP požadavky zasílají jako klasické HTTP požadavky. Samozřejmě – je na straně serveru, jak se s tímto chováním smíří a zda vrácené výstupy budou dávat tytéž výsledky, jako u plné AJAXové podpory.

9.1.5 Shrnutí

Výhody:

Hlavní výhoda tkví v urychlení práce, protože se nemusí pokaždé načítat nová stránka. Toto chování je daleko blíže tomu, co zná uživatel z klasických desktopových aplikací. Jedním z pravidel dobré použitelnosti je držet se toho, co už uživatel zná. AJAX také většinou šetří datové přenosy. U klasické webové aplikace se s každým požadavkem musí uživateli posílat celý kód stránky, který neobsahuje příliš mnoho nových a důležitých informací. Naopak s technologií AJAX se posílá jenom to důležité. Ve spolupráci s JavaScriptem tak lze vytvářet dynamické, rychle reagující stránky na požadavky uživatele.

Nevýhody:

AJAX znemožňuje použití tlačítka zpět v prohlížeči, protože to lze použít pouze pro statické stránky (pamatuje si klasické http požadavky). Toto se dá bez váhání označit za největší problém technologie AJAX. Uživatelé jsou na tlačítko zpět zvyklí a očekávají od něj určitou funkci. Při použití technologie AJAX však toto tlačítko vrátí uživatele na předcházející stránku, což ovšem neznamená návrat aplikace do předcházejícího stavu. Při změnách na stránce pomocí technologie AJAX se nemění URL v adresním řádku prohlížeče. Proto není možné takto modifikovanou stránku poslat e-mailem nebo uložit do záložek (lze řešit pomocí speciálních JavaScriptů). AJAX neřeší vše. Je stále pouze nadstavbou nad stávajícími webovými technologiemi, která se snaží překonat některá jejich omezení. Protokol http vůbec není vhodný pro aplikace spolupracující intenzivně se serverem – problémem je, že se při každém požadavku musí navázat spojení se serverem, které se po jeho vyřízení ukončí. Tímto může dojít ke zpomalení aplikace. Při nevhodném použití může také zvýšit datový přenost. AJAX také nemusí vždy fungovat – jelikož je založen na JavaScriptu, pokud jej uživatel zakáže v prohlížeči, tak celá technologie přestává fungovat.

Kontrolní otázky:

1. Co je to RIA?
2. Co je to AJAX?
3. Jaká je odlišnost chování webové stránky při implementaci pomocí AJAXu?

Úkoly k zamyšlení:

V kapitole byl uveden základní rozdíl mezi tvorbou běžné webové aplikace a aplikací tvořených pomocí technologie AJAX. Zamyslete se nad budoucími směry tvorby webových aplikací, nad výhodami použití technologie AJAX. Vraťte se k tomuto úkolu znovu po prostudování kapitoly o webových službách.

Korespondenční úkol:
Nalezněte na internetu příklad webových stránek, které používají technologii AJAX. Porovnejte je se stránkami, které komunikují klasicky a proveďte srovnání. Rozhodnutí zdůvodněte.

Shrnutí obsahu kapitoly
V této kapitole byla představena technologie AJAX a ukázána tvorba aplikace s její pomocí. Byly ukázány požadavky na běh AJAXových aplikací a ukázána tvorba jednoduché aplikace využívající technologii AJAX.

10 Pokročilý návrh v UML

V této kapitole se dozvíte:

- Jak přesněji pracovat s vybranými diagramy.
- Jak přiblížit procesu návrhu do procesu implementace v UML diagramech
- Jak obecně používat ostatní diagramy s využitím konvencí pojmenování a stereotypů.

Po jejím prostudování byste měli být schopni:

- Korektně namodelovat danou problematiku ve vybraných diagramech.
- Rychle pochopit princip a způsob práce s dalšími diagramy UML.

Klíčová slova této kapitoly:

UML

Doba potřebná ke studiu: 3 hodiny

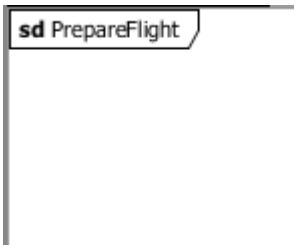
Průvodce studiem

Kapitola představuje problematiku modelování pokročilejších požadavků a a jejich konstrukcí do vybraných UML diagramů. Představuje také obecné pojmy týkající se UML diagramů.
Ke studiu se doporučuje vyzkoušet si ukazované příklady ve vlastním, vhodně zvoleném modelovacím nástroji.
Na studium této části si vyhraďte 3 hodiny.

Problematika UML diagramů byla již částečně představena ve studijní opoře pro předmět Informační systémy 1. Protože spolu se získanými znalostmi je třeba vědět, jak korektně zapisovat i pokročilejší konstrukce v UML diagramech a získat obecnější přehled o jejich fungování, některé principy a vybrané diagramy zde budou představeny blíže. Jedná se ale stále o obecný přehled, zájemce o bližší studium odkazujeme například na [IBM1, IBM2, IBM3, IBM4, Gr], pro podrobnější informace pak na [Pe].

10.1 Diagramy

Každý diagram v UML je podle standardu 2.0 je reprezentován rámcem, který vlevo nahoře v malém čtverečku obsahuje zkratku typu diagramu a název diagramu.
Rámce jsou obecnou strukturou UML, která umožňuje zahrnovat v sobě jiné artefakty a v levém horním rohu vždy obsahuje název nebo očekávaný popis rámce.

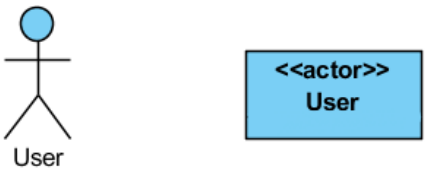


Prefix **sd** říká, že tento diagram je sekvenční. Každý z diagramů v UML má svou vlastní zkratku. Za prefixem následuje název diagramu.
Obecně se však jedná o rámec, který lze použít kdekoliv pro „zabalení“ skupiny artefaktů, které k sobě patří. Tehdy se samozřejmě prefix typu diagramu vynechává a vkládá se pouze název rámce. S takto představenými rámci se setkáme i v dalších diagramech.

10.2 Stereotypy

Stereotyp je technika, která v UML slouží k upřesnění významu artefaků. Cílem je blíže specifikovat chování daného objektu, nebo jeho vlastnosti. V UML se diagramy zakreslují pomocí artefaktů – značek, které reprezentují zachycované objekty. UML pro každý artefakt definuje obecný vzhled, jak má vypadat. Někdy ale designer/analytik potřebuje zachytit určité odlišnosti, fakta, specifika, které chování daného artefaktu mírně upravují nebo odlišují od běžného chování. Toto se provádí pomocí tzv. stereotypů.
Stereotypy se zakreslují typicky do artefaktu nahoru nad název popisující daný artefakt, nebo nad daný artefakt (typicky u čar), a to do dvojice lomených závorek: <<název_sterotypu>>. Obecně použití stereotypu není omezeno, ale jejich nadměrné používání může diagramy značně znepřehlednit. Z toho důvodu, že určité stereotypy se používají často, byly nahrazeny zvláštním symbolem artefaktu, který sám o sobě stereotyp reprezentuje. Tehdy je samozřejmě důležité používat tyto nahrazující grafické artefakty, z důvodu jejich zavedenosti a snadné čitelnosti. Nejběžnějšími příklady stereotypů a jejich nahrazení uvádí následující obrázky.

Kurzíva v názvu typicky značí abstraktní objekt. Můžeme jej ale také zapsat běžným fontem, a doplnit stereotyp *abstract*.



Postavička v UML značí aktora. Můžeme ale použít obecný tvar a připojit stereotyp *actor*.

10.3 Identifikace objektů

V diagramech UML typicky potřebujete identifikovat artefakty k nejrůznějším objektům – třídy potřebují své názvy, stejně jako aktoři, případy užití a další. V rámci UML 2.X se můžete odkazovat třemi různými způsoby. Následující příklady budou uvažovat použití v diagramu tříd (protože je nejsnáze pochopitelný i pro začínající OOP programátory a UML designéry), ale lze je samozřejmě využít obecně v libovolném diagramu, kde toto použití bude dávat smysl.
Prvním způsobem je klasický název:

```
Flight
```

Takto uvedeno se jedná o obecnou šablonu skupiny objektů, nejblíže asi pojmu „název třídy“. Jedná se o klasický text (nepodtržený, ne kurzívou) reprezentující název. V diagramu tříd tedy budeme vědět, že artefakt popisuje třídu nazvanou *Flight*.
Druhým způsobem je odkaz na instanci:

```
EZY5495
```

Podtržený text znamená, že se nejedná o šablonu (tedy ne třída), ale přímo o jednu konkrétní instanci. Tato jedna konkrétní instance má své **jedinečné jméno**. Je-li třeba (a pokud je to známo), lze za dvojtečku doplnit název šablony (třídy), ke které se instance vztahuje.[29] Pro zdůraznění se doplňuje dvojtečka „za“ i v případě, že třída není známa (její název se pak nepíše). Po doplnění bude zápis vypadat například:

```
EZY5495 :
EZY5495 : Flight
```

Poslední variantou je tzv. „role“. Role reprezentuje skupinu instancí stejného chování, ale neodkazuje nutně na konkrétní objekt. Od zápisu instance se liší nepoužitím **podtržení**. Abychom byly schopni odlišit role od tříd, u rolí vždy za název doplníme dvojtečku, i přesto, že nevíme, jaký název třídy za roli uvést. Následují příklady rolí reprezentující „lety společnosti easyJet s označením letu začínající na EZY“:

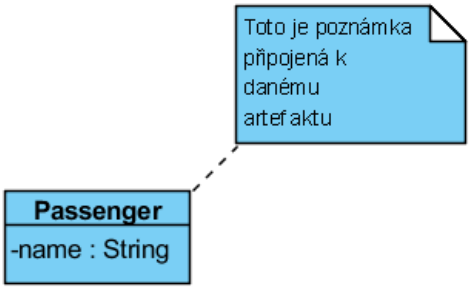
```
EZYFligth :
EZYFlight : Flight
```

Pro doplnění – někteří návrháři naopak pro zdůraznění, že určitý název není rolí, ale přímo typem, přidávají dvojtečku „před“ zápis:

```
: Flight
```

10.4 Obecné poznámky

Někdy je třeba do UML zapsat informaci, která nejde klasickým způsobem pomocí standardních prvků zaznačit, ale její prezentace je důležitá. Tehdy lze využít obecného mechanismu poznámek, který umožňuje k libovolnému artefaktu pomocí přerušované čáry připojit obdélník s přehnutým rožkem, do kterého je možno vložit libovolný text.



[29] Není to však nutné. Zejména v úvodních fázích návrhu může analytik chtít specifikovat, že existuje konkrétní instance, které bude mít určitý stav nebo určité chování, ale ještě nemůže vědět, do jaké třídy (zda vůbec) bude patřit.

Připojení poznámky má úplnou volnost – lze ji připojit k jakémukoliv jinému grafickému symbolu ve všech UML diagramech, měly by se však **používat co nejméně**, protože:

- Značně snižují přehlednost UML diagramu, zvláště, pokud se jich v daném diagramu vyskytuje větší množství;
- Jsou neformálním mechanismem. Zatímco ostatní stavební bloky jsou formální a dají se tedy automatizovaně zpracovávat (a převádět například na zdrojový kód ve vybraném jazyce), poznámky formálně zpracovat nelze, protože můžou obsahovat cokoliv.

Cílem tedy je použití poznámek minimalizovat a nahrazovat je jinými, standardními a formálními přístupy, nejlépe stereotypy.

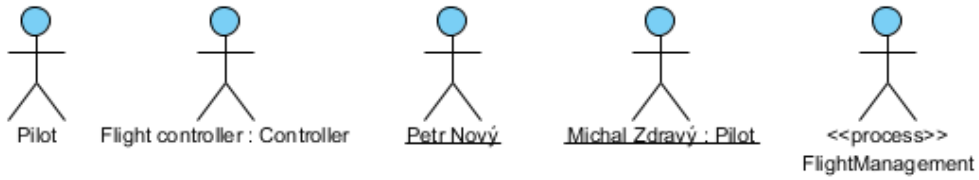
10.5 Diagram případů užití – Use case diagram

Diagram případů užití již byl poměrně podrobně představen v předchozí studijní opoře věnované předmětu KIP/INFS2. Tedy jenom stručně: případy užití ukazují funkcionalitu nabízenou systémem. Hlavním cílem je vizualizovat tuto funkcionalitu vzhledem ke tzv. aktorům – typicky lidským uživatelům, a také zachytit vazby mezi jednotlivými případy užití.

Důležité oblasti, které je třeba si zapamatovat při tvorbě obecného diagramu případu užití:

- Přestože **aktor** bývá typicky graficky reprezentován symbolem postavičky, a některé zdroje uvádějí, že se jedná o lidského uživatele, obecná definice říká, že „aktor je subjekt, který je v nějakém vztahu komunikace se systémem“. **Aktor tedy vůbec nemusí být člověkem**, ale může jím být například webová služba, nebo jiný systém, který bude využívat služeb modelovaného systému. Aktor také nemusí aktivně vyvolávat komunikaci, ale může být konzumentem služeb vyžadovaných modelovaným systémem – například, při modelování systému bankomatu bude tento bankomat muset komunikovat s bankou. Banka, ač sama aktivně komunikaci s bankomatem nevyvolá (vždy je bankomatem zavolána), bude i v tomto případě v systému aktorem. Typicky tedy aktorem bývá uživatel, organizace, stroj nebo jiný externí systém.
- V každém UC diagramu je velmi nutné modelovat **hranice systému**, typicky jako obdélníkové orámování UC, které do systému patří. Čtenář diagramu bude okamžitě vědět, které věci jsou součástí systému a které jsou již vně.
- Vzhledem k vztahům mezi případy užití (vztahy budou uvedeno dále) je důležité hledat a analyzovat případy užití z pohledu jejich užitelnosti. V diagramu by se neměl vyskytovat případ užití, který není spojen s žádným aktorem a ve vztahu s některým z ostatních případů užití. Stejně tak aktor, který nevyvolává žádný případ užití, není aktorem vůči modelovanému systému.

U aktorů můžeme využívat identifikaci objektů, jak bylo uvedeno výše.

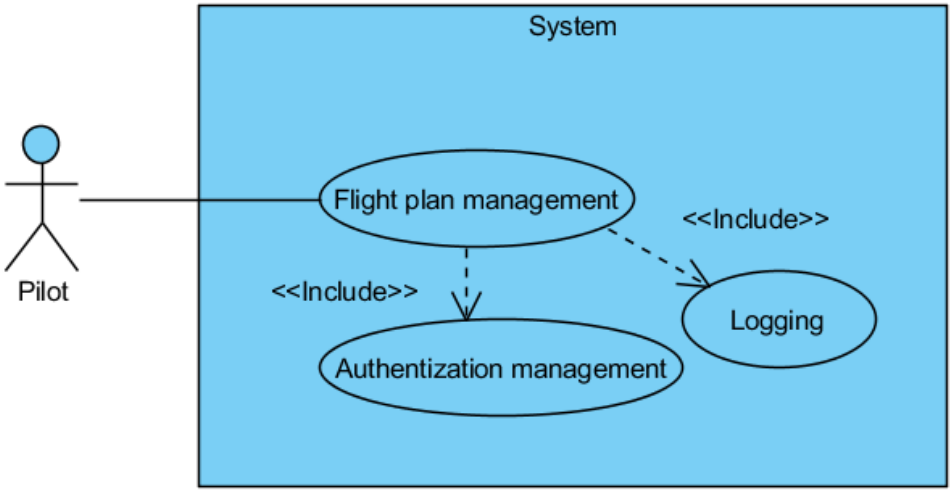


Pilot je obecným aktorem. Flight Controller je nějaký letový řídící (ne konkrétní instance, ale obecná role) – navíc instance této role je povinně řídícím. Petr Nový je konkrétní instance – aktor. Michal Zdravý je také konkrétní instance – aktor a navíc je pilotem. FlightManagement je aktorem, není to ale osoba, nýbrž nějaký vykonávaný proces.

V některých nástrojích lze specifikovat i obecný tvar pro aktora (například obdélník), jinak je nutno explicitně (ideálně stereotypem) určit, o jaký typ aktora se jedná.
Aktoři se se svými případy užití (tj. s případy, které používají, nebo kterými jsou využíváni) spojují plnou čarou. Je-li třeba explicitně naznačit jednosměrnou komunikaci, může být čára doplněna šipkou.
Kromě vztahu „aktor – případ užití“ mohou vznikat vztahy i mezi aktory a stejně tak mezi případy užití.

10.5.1 Vnoření případů užití – include

„Vnoření“ případů užití je jedním z nejčastějších a nejběžnějších vztahů mezi případy užití. Závislost *include* od prvního případu užití (nazývaného také *base use case*, dále pojmenovaného jako A) k druhému případu užití (*inclusion use case*, dále pojmenovaného jako B*)* značí, že **první případ užití A bude zahrnovat (nebo volat) druhý případ užití B**. Jeden případ užití může mít vnoření do libovolného počtu dalších případů užití. Vnoření se zaznačí přerušovanou šipkou s přidaným stereotypem <<include>>. Případ užití A je se svým popisem zodpovědný za korektní určení „kdy a jak“ bude daný vnořený případ užití B volán. Tento popis ale již typicky není součástí UML diagramu, ale značí se do textové části popisu projektu.

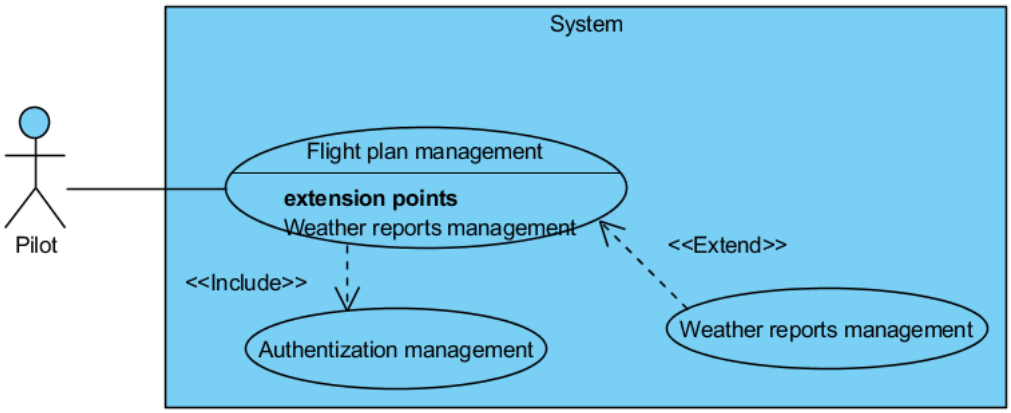


Ve výše uvedeném případě může pilot pracovat se správou letových plánů. Před práci s ní se ale typicky bude muset autentizovat (povinně) a práce s letovými plány se bude také monitorovat do logů.
Důležité je uvědomit si vztahy obou případů užití. Případ užití A uvnitř **nutně potřebuje vyvolat** případ užití B, a tudíž o případu užití B ví. Samostatně bez

něj nemůže fungovat. Oproti tomu případ užití B nic neví o případu užití A. Typicky, vnořované případy užití (v našem případu B) nejsou schopné fungovat samostatně. **Typickým použitím *include* je vytknutí společných funkcionalit** do nového případu užití, který si budou původní případy užití připojovat.

10.5.2 Rozšíření případů užití – extends

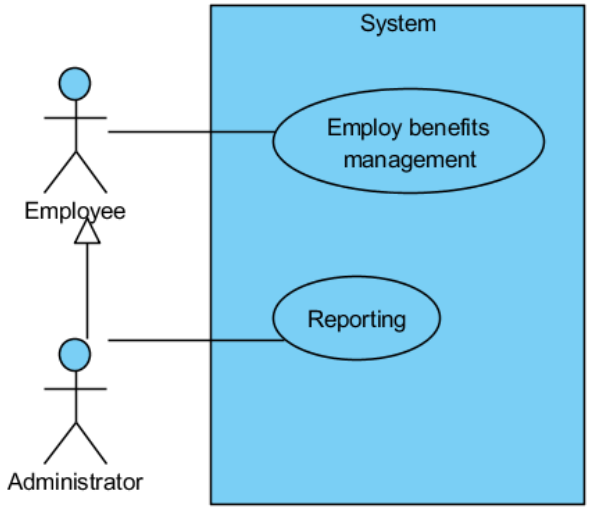
Rozšíření je další varianta vztahu případu užití. Tentokráte máme první případ užití (nazývaný *extensit use case*, dále označený jako B (!)), který rozšiřuje původní případ užití (nazývaný *base use case*, dále označený jako A). Důležité je povšimnout si **opačné orientace vztahu** oproti variantě *include*.
Rozšíření umožňuje rozšířit případ užití A o nové funkcionality reprezentované případem užití B, které původní případ užití neobsahuje. Do definice případu užití A se zavádějí tzv. *extension points* (body rozšíření), které říkají, že v určitých místech/fázích případu užití A lze jeho chování rozšířit.



Výše uvedený příklad říká, že případ užití *Flight plan management* může být rozšířen o část umožňující zjišťovat a pracovat s informacemi o počasí. Toto rozšíření není povinné – pokud modul správy počasí nebude existovat, pilot může pořád pracovat se správou letových plánů, jenom si informace o počasí bude muset zjistit někde jinde.
Závislost případů užití je zde tedy **opačná** oproti *include*. Základní (rozšiřovaný) případ užití A neví nic o rozšiřujícím případu užití B (vyjma *extension points*) a **je schopen plně fungovat i bez jeho existence**. Případ užití B volitelně rozšiřuje případ užití A a typicky o jeho existenci ví a odkazuje se na něj. Opět, rozšiřující případ užití B není typicky schopen fungovat samostatně.

10.5.3 Dědičnost mezi aktory

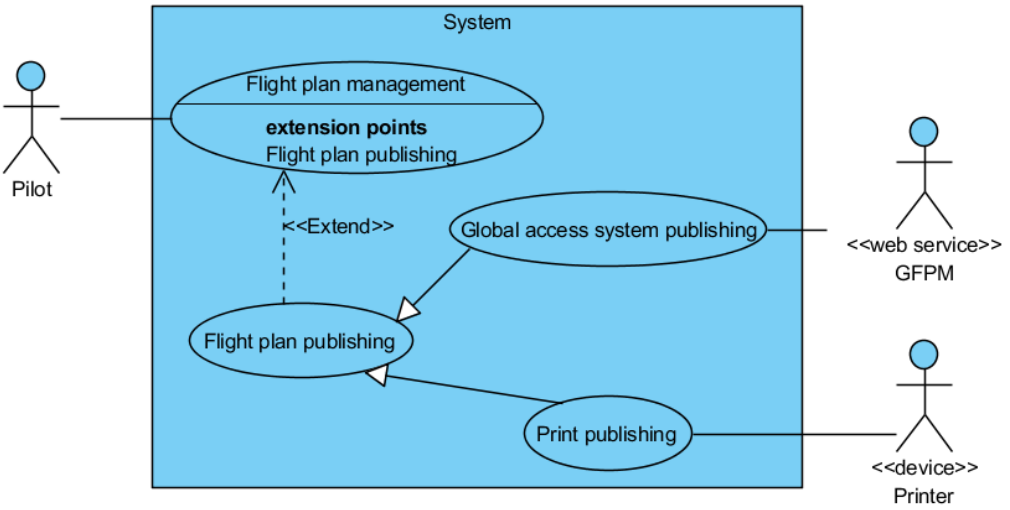
Dědičnost lze chápat jako klasický vztah generalizace/specializace, kdy chceme při modelování naznačit, že určitý aktor (potomek) sdílí všechno, co „umí“ jiný aktor (předek) a navíc může mít své vlastní další schopnosti. Potomek má dostup na všechny případy užití předka, navíc může mít své případy užití, na které se předek nedostane.



Zaměstnanec může pracovat v systému se zaměstnaneckými benefity. Administrátor může totéž, navíc však může nahlížet do sestav a reportů.

10.5.4 Dědičnost mezi případy užití

Dědičnost mezi případy užití je poměrně složitější v případě složitých nebo nevhodně namodelovaných případů užití. Opět, principem je získat od případu užití (předka) do případu užití (potomka) dědit aktory, sekvence chování a body rozšíření (*extensit points*, bude vysvětleno dále). U složitějších případů užití se dědičnost používá hlavně ke zpřehlednění aktorů, které budou daný případ užití používat.



Výše uvedený příklad ukazuje, že letové plány lze obecně publikovat – publikování konkrétně bude umožňovat systém buď do globální celosvětové databáze letových plánů, nebo na lokální tiskárnu.

10.6 Diagram aktivit – Activity diagram

Diagram aktivit v UML je základní, poměrně jednoduchý diagram, určený k zachycení chování které závisí na výsledcích vnitřních procesů. Diagram aktivit je reprezentován tokem mezi očekávanými operacemi (aktivitami) – k další aktivitě se přechází po splnění aktivity předchozí. Diagramy aktivit jsou vhodné při definici operací jako doplnění pro diagramy případů užití, kdy

umožňují zjistit a definovat toky dat, které mezi sebou propojují právě jednotlivé případy užití.
Každý diagram aktivit se skládá minimálně ze 3 základních bloků:

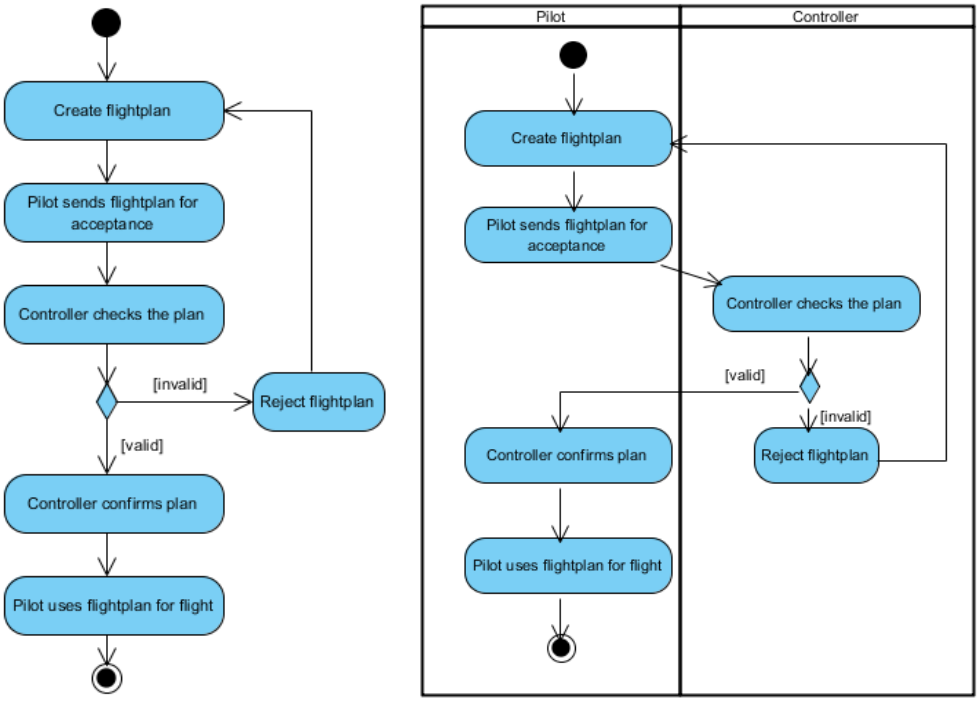
- Počáteční uzel – reprezentován černým kolečkem;
- Aktivitou (akcí) – reprezentován obdélníkem s oblými rohy a popisem akce;
- Koncovým uzlem – reprezentován černým kolečkem s černým orámováním.

Tyto bloky jsou mezi sebou spojeny orientovanou jednosměrnou šipkou.

Počáteční uzel říká, kde aktivita vzniká. Aktivity popisují jednotlivé akce, které se v průběhu diagramu aktivit dějí. Koncový uzel značí konec běhu aktivity v diagramu. Diagram může obsahovat maximálně jeden počáteční uzel, ale může obsahovat více koncových uzlů. Z bloku akce nemůže vycházet více než jedna odchozí šipka (popis by potom nebyl deterministický).

Klasickým blokem typickým v diagramu aktivit je rozhodování. Místo rozhodnutí je reprezentováno kosočtvercem, ze kterého může vycházet více odchozích hran. U každé hrany se do složených závorek zapisuje podmínka, která má být splněna, aby se danou hranou mohlo pokračovat.

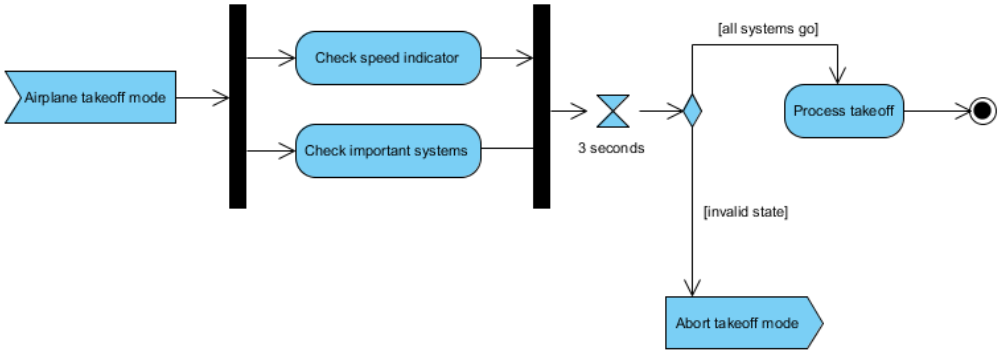
Další volbou zpřehledňující diagram aktivit jsou tzv. „sim-lanes“ – plavební dráhy. U složitějších diagramů, nebo je-li to třeba, lze definovat, které konkrétní objekty jsou s danou aktivitou spojeny. V diagramu se následně vymezí obdélníkové prostory, kam se její aktivity zapisují. Představené formalismy ukazují následující obrázky.



Jen ve stručnosti další stavební bloky, které lze použít:

- Paralelní vykonání lze zachytit uzavřením bloku aktivity do dvou svislých čar.

- Je-li třeba zdůraznit časové hledisko, lze použít artefakt přesýpacích hodin s doplněním časového intervalu.
- Další běžnou oblastí jsou tzv. signály. Aktivity nemusí nutně začínat počátečním uzlem – aktivitu může vyvolat i vnější vstup – tzv. signál (událost). Zakresluje se obdélníkem s vnořenou stranou. Diagram aktivit může obsahovat několik signálů, které mohou do procesu diagramu aktivit vstoupit. Obdobně, aktivita může vyvolat výstupní signál (který se předává dále). Výstupní signál se zakresluje obdélníkem s vystouplou stranou (viz následující obrázek).
- Speciálním případem hrany je zalomená šipka do tvaru blesku. Taková šipka upozorňuje na nestandardní výjímečné volání způsobené typicky chybou (tzv. výjimky).



10.7 Diagram tříd – Class diagram

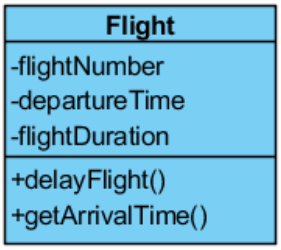
Diagram tříd je, jak již bylo představeno dříve, základním diagramem pro reprezentaci typů, které budou modelovány ve vytvářeném systému. Typicky se modelují čtyři základní definice: třídy, rozhraní, datové typy[30] a komponenty. Obecně budou nejdříve příklady ukázány na diagramu tříd, na případné odchylky bude ukázáno později.
Na začátek trochu opakování.

10.7.1 Základní modelování třídy

Základní reprezentace třídy je obdélník. Nutně musí obsahovat tučný text definující název třídy.

V pokročilejších fázích návrhu již potřebujeme do třídy doplnit budoucí modelované členy tříd. Reprezentace třídy se změní na obdélník rozdělený horizontálně na tři části. V horní části se nachází pojmenování. Prostřední část obsahuje názvy třídních proměnných, v nejnižší části se nacházejí metody. I pokud ještě nejsou známy parametry metod, k názvům metod se doplňují (prázdné) kulaté závorky.

[30] Ve smyslu obecnějšího pojmu, než třída, například referenční, hodnotové, výčtové aj, dle zvolené technologie.



Na výše uvedeném obrázku již lze vidět ještě jeden doplněný artefakt – jedná se o definici viditelnosti. UML umožňuje modelovat základní viditelnost pomocí symbolů:

- + public
- # protected
- - private
- ~ package

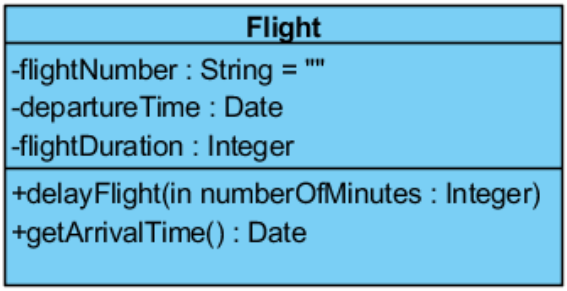
V pokročilejší fázi návrhu již můžeme doplnit další důležité poznámky k chování třídy. Zejména se samozřejmě doplňují datové typy, parametry funkcí, návratové hodnoty a výchozí hodnoty. Obecná syntax pro úplný zápis třídní proměnné může vypadat:

```
<visibility> <name> : <dataType> = <defaultValue>
```

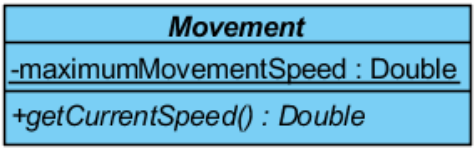
Obdobně u funkcí můžeme zapsat obecnou syntax – všimněte si, že u jazyků, které to podporují, můžeme explicitně ujasnit chování parametrů (vstupní, výstupní, explicitně předávané referencí apod.):

```
<visibility> <name> (<in|out|ref> parameterName :
parameterType) : returnType
```

Návratový typ funkce doplňujeme pouze tehdy, pokud není *void*.

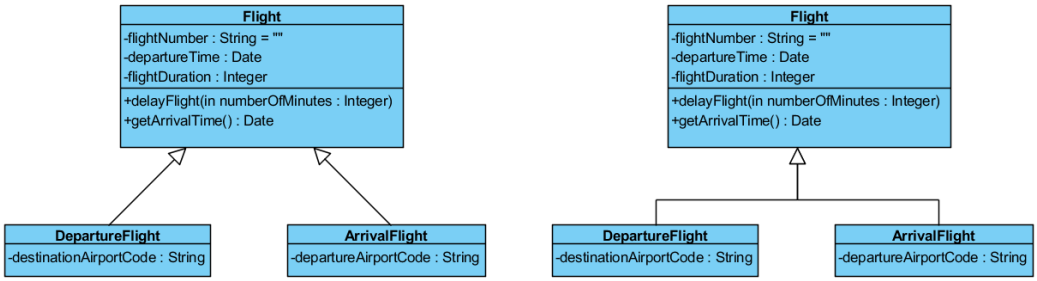


Pokud chceme vyjádřit, že třída nebo některý z jejich členů je abstraktní, zapíšeme jej pomocí *kurzívy*. Pokud chceme ve třídě zmínit statického člena, podtrhneme jej.



10.7.2 Vztahy mezi třídami – dědičnost

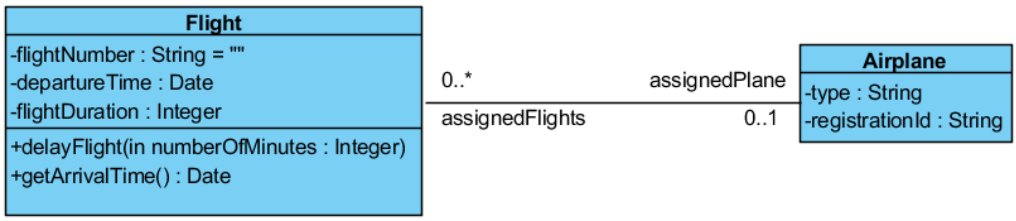
Základním vztahem mezi třídami je dědičnost. Zakresluje se pomocí šipky s plnou čarou a prázdnou špičkou. Je možno využít šipky přímé, i zalomené do stromu (které jsou typicky přehlednější).



10.7.3 Vztahy mezi třídami – asociace, agregace, kompozice

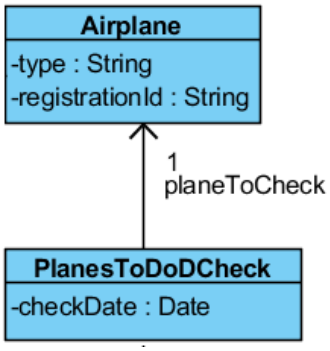
Problematika asociací je poměrně rozsáhlá, budou proto představeny alespoň ty nejzákladnější varianty.

Základní variantou je obecná asociace identifikující nějaký vztah mezi dvěma třídami. Obě třídy vědí, že jsou ve vztahu s jinou třídou. Zakresluje se pomocí plné čáry jako spojnice, doplňují se povinně názvy asociací a typicky kardinalita.



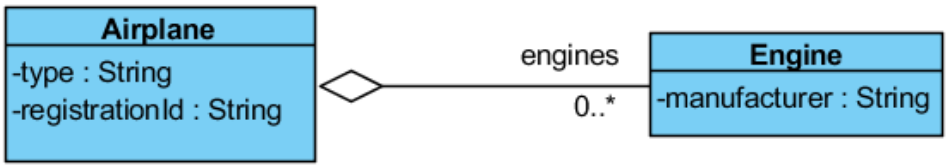
Názvy asociací se zapisují **napříč** k druhé třídě, tedy v našem případě třída *Flight* bude mít jeden z atributů *assignedPlane* a naopak třída *Airplane* bude mít třídní proměnnou *assignedFlights*.

Dalším případem je jednosměrná asociace – tehdy jedna ze tříd neví, že je ve vztahu s druhou třídou.

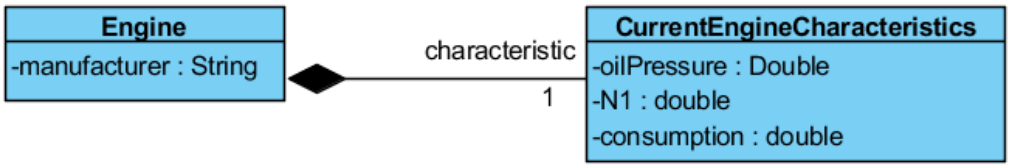


Název asociace a kardinalita se tedy nastavuje pouze u jedné strany asociace, čára je navíc typicky doplněna šipkou (případně kardinalitou 0), zápis se opět provádí napříč.

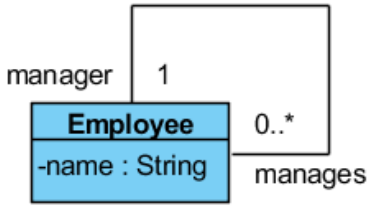
Silnější variantou asociace je **agregace**. Agregace naznačuje těsnější vztah mezi objekty, a zapisuje se pomocí kosočtverce bez výplně u nadřazeného objektu. Typicky se opět doplňuje o kardinalitu. Jedná se však stále o volnou vazbu, takže objekty mohou smysluplně existovat bez nadřazeného objektu – v našem případu motor patří k letadlu, ale lze jej demontovat a může existovat samostatně bez letadla.



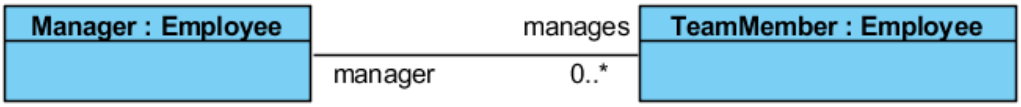
Poslední základní variantou asociace je **kompozice**. Ta se zakresluje pomocí kosočtverce s výplní a používá se v případě, kdy podřízený objekt nemůže smysluplně existovat bez svého nadřazeného objektu – v našem případě popisné charakteristiky motoru nemohou existovat bez motoru, který popisují[31].



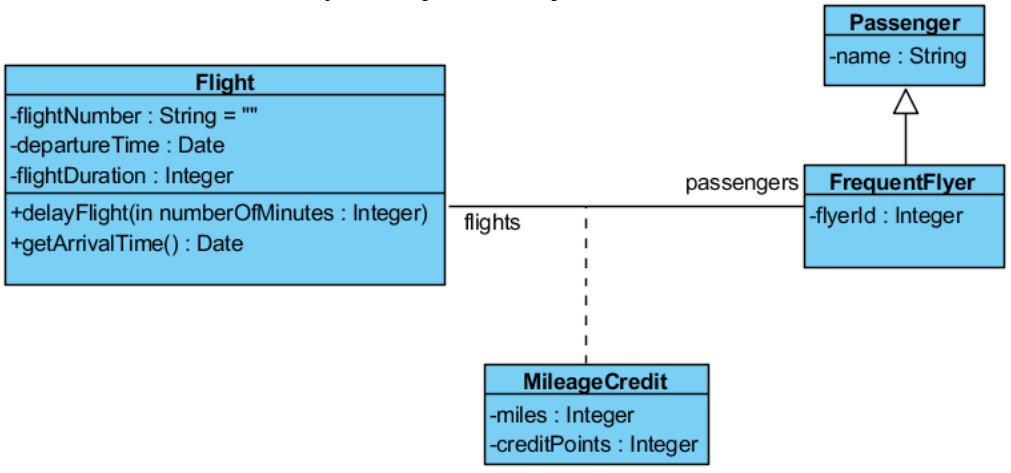
Další běžný modelovaný příklad jsou reflexivní asociace, kdy třída odkazuje sama na sebe. Jejich nevýhodou je nízká přehlednost.



Lze si povšimnout, že tento zápis není moc přehledný. Mnohem vhodnější je použít rolí (viz předchozí kapitola) a rozepsat jej pomocí vztahu mezi dvěma rolemi třídy *Emplyee*.



Názvy nejsou podtrženy, jedná se tedy o role a nikoliv o instance třídy.
Poslední ukázanou vlastností budou asociační třídy. Používají se v případech, kdy je třeba dekomponovat asociace M:N, nebo pokud potřebujeme k asociaci dodat nějaké atributy, které ani jednomu z asociovaných objektů nepřísluší. Vizualizaci asociační třídy ukazuje následující obrázek.

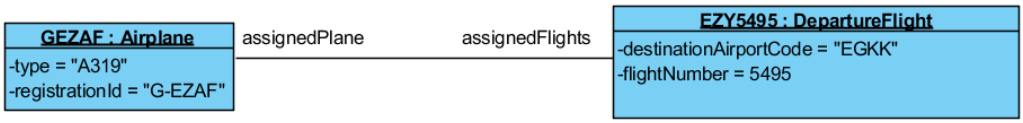


10.7.4 Reprezentace chování instancí

V určitých případech je potřebné ukázat na určité vlastnosti konkrétních rolí nebo objektů daných tříd. Tehdy se UML řídí výše uvedenými principy – pokud chceme naznačit, že informace uvedené v diagramu patří ke konkrétní instanci, potrhneme název instance (tam kde se původně vyskytoval název

[31] Jiným názorným příkladem je firma a její oddělení. I zde se jedná o kompozici, protože pokud firma zanikne, oddělení nemohou existovat dále.

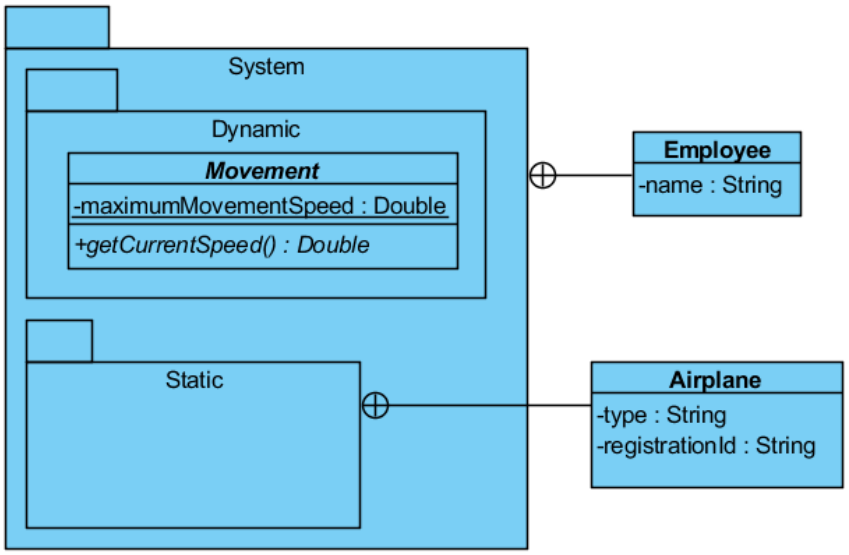
třídy) a doplníme i typ třídy, od které je instance odvozena. V dalších blocích, kde jsou uvedeny očekávané hodnoty třídních proměnných. **Doplňujeme pouze důležité hodnoty** – nedoplňují se hodnoty atributů, které nejsou známé, nebo důležité, přestože u obecného popisu třídy se vyskytují. Kombinací lze naznačit i vztahy mezi instancemi tříd, jak ukazuje následující obrázek.



Tato technika funguje až v UML 2.0 a ne všechny editory ji umí korektně zaznačit. V případě problémů je nutno využít stereotypů nebo poznámek.

10.7.5 Packages

Na závěr pouze stručně – kvůli přehlednosti se samozřejmě dají jednotlivé třídy umísťovat do balíčků, které lze rekurzivně zanořovat. Typicky lze použít dva přístupy – první spočívá v uvedení tříd dovnitř artefaktu reprezentujícího balíček – příkladem bude na níže uvedeném obrázku třída *Movement* v balíčku *System.Dynamic*. Druhou variantou je odkazování pomocí spojnic doplněných znakem „+“ v kolečku – příkladem je třída *Employee* v balíčku *System* nebo třída *Airplane* v balíčku *System.Static*. Obě formy zápisu jsou si ekvivalentní.

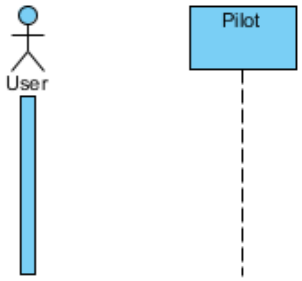


10.8 Sekvenční diagram

Dalším důležitým diagramem je sekvenční diagram, zachycující interakci mezi objekty v daném pořadí. Sekvenční diagram typicky v úvodních fázích návrhu zahrnuje obecné objekty a obecné postupy jejich komunikace, spolu s upřesňováním návrhu a prováděním implementace se upravují nebo transformují do diagramu popisujících typicky konkrétní třídy nebo objekty odpovědné za provádění určitých operací, a jejich metod.

10.8.1 Život objektu

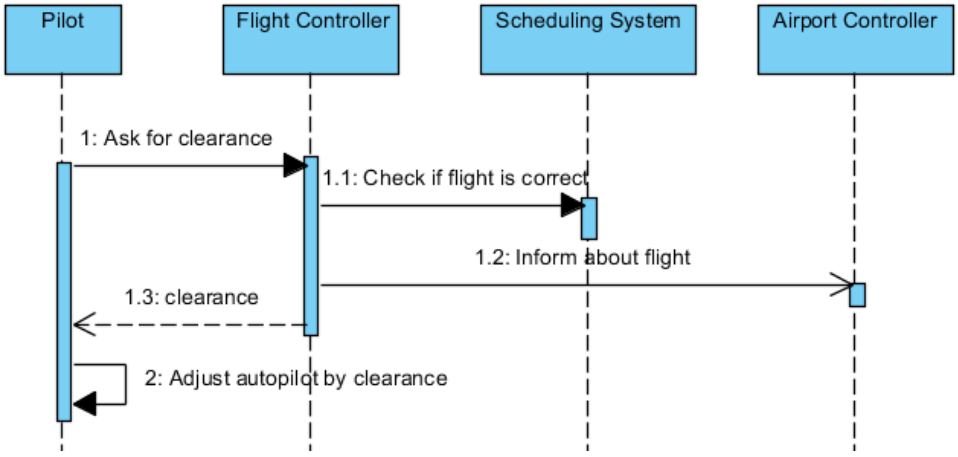
Jak bylo zmíněno, sekvenční diagram zachycuje objekt a jeho interakci s okolím. Proto má každý objekt v rámci sekvenčního diagramu (kromě svého názvu) i tzv. životní čáru. Ta se zakresluje pod objekt přerušovanou čarou a její existence reprezentuje **existenci konkrétní instance** vykonávající daný sekvenční diagram.



V záhlaví je opět uveden název objektu (třídy/ instance/role, dle pravidel uvedených na začátku kapitoly). Přerušovaná čára reprezentuje konkrétní instanci dané třídy. Objektem sekvenčního diagramu nemusí být něco, co si návrhář představuje jako třídu, ale samozřejmě také aktoři i jiné stavební prvky (databáze, rozhraní, webová služba apod.).

10.8.2 Zasílání zpráv

Cílem sekvenčního diagramu je reprezentovat zasílání zpráv. Proto typicky objekty interagují mezi sebou.
Základním mechanismem je zaslání zprávy. Zpráva má zdroj a cíl, zakresluje se typicky vodorovnou šipkou, nebo šipkou s plnou výplní.
Volání vytvoří nad životní linií přijímacího objektu modrý svislý pruh – tento pruh reprezentuje zpracovávání přijímaného požadavku (nereprezentuje tedy existenci instance)[32].



Ve výše uvedeném příkladu tedy jednotlivé body znamenají:

1	Pilot aktivně požádá letového řídícího o udělení povolení k provedení letu
1.1	Letový řídící aktivně zkontroluje přes objekt plánovače letů, zda je let připraven v pořádku.
1.2	Letový řídící informuje o budoucím letu letištního řídícího.
1.3	Letový řídící zašle zpět pilotovi povolení k letu.
2	Pilot zapíše získané povolení do počítače letadla.

Důležité poznámky:

- Nad šipky reprezentující komunikaci se dopisuje popisný text, který říká, jaká operace se provádí (jaká zpráva se zasílá). Opět, v úvodních

[32] Zjednodušeně si to lze programátorsky představit jako blok volané funkce { }, když je zpracováván. Toto přirovnání ale nemusí platit vždy.

fázích návrhu se může jednat o obecný text, později tyto texty typicky reprezentují volání metod volaných objektů. Volání se běžně pro přehlednost i číslují.

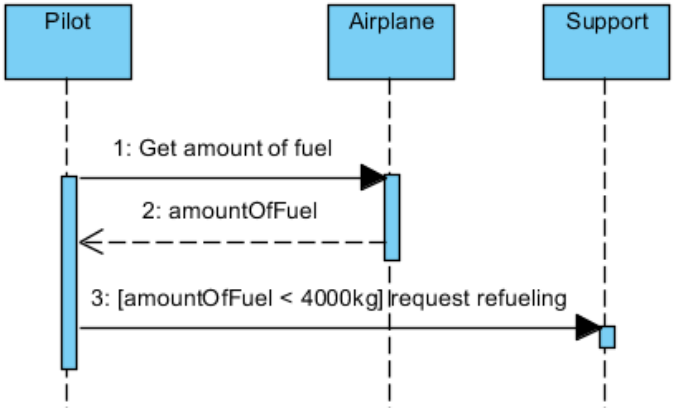
- Z pohledu pilota se jedná o jeden požadavek a **čekání na výsledek**. Bez výsledku pilot neví, zda může pokračovat. Protože se čeká na vyhodnocení volání, jedná se o **synchronní volání**. Tato volání jsou reprezentována plnou šipkou u volaného objektu. Synchronní volání je i volání 1.1.
- Vrácení informace o výsledku pilotovi se provádí v bodu 1.4 pomocí **přerušované** šipky vedoucí zpět k objektu, který interakci vyvolal. Tato zpětná šipka se ale zakresluje pouze tehdy, pokud chceme explicitně zdůraznit, že je volání ukončeno, nebo upozornit na vracenou hodnotu (kterou potom můžeme dále v diagramu používat). Velké množství těchto zpětných šipek značně zesložiťuje diagram. Ve volání 1 explicitně upozorňujeme na vracení objektu *clearance*, který dále můžeme využít. Oproti tomu u volání 1.1 nepožadujeme explicitně zpět navrácení hodnoty (můžeme například předpokládat, že v případě problému celý sekvenční diagram skončí chybou) a zpětnou přerušovanou šipku tedy neděláme.
- Volání 1.2 se provádí **asynchronně** – letový řídící informuje letištního řídícího o budoucím letu, ale nečeká dále na jeho odpověď a hned provádí další operace (zde 1.3). Šipka proto nekončí plným tvarem. V těchto případech samozřejmě nemá smysl kreslit navrácení hodnoty, protože nevíme, kdy se tato hodnota vrátí (za jak dlouho řídící letiště na zaslanou informaci zareaguje). Povšimněte si ještě jednou odlišností mezi synchronním a asynchronním voláním. U synchronního volání 1 pilot čeká na vrácení hodnoty 1.3 (i kdyby tam tato šipka nebyla, čeká na uběhnutí zpracování požadavku 1) a teprve potom pokračuje bodem 2. U asynchronního volání by pilot nečekal, odeslal by informaci letovému řídícímu a hned by pokračoval bodem 2 (což by samozřejmě nedávalo smysl). Je proto důležité tyto typy volání odlišovat.
- Bod 2 reprezentuje volání sebe sama – pilot může zaslat zprávu i sám sobě. Je-li třeba demonstrovat rekurzivní volání, vytvoří se u dané šipky další životní linie objektu (objekt má pak dvě překrývající se životní linie).

Tyto základy typicky stačí k modelování sekvenčního diagramu pro základní scénáře v brzkých fázích projektu. Postupně, jak se projekt a jeho řešení stává více konkrétním, potřebuje návrhář zachytit více a více programátorské konstrukce, které blíže objasňují, jak se bude daný kód provádět, jako jsou podmínky, cykly a další.

10.8.3 Guards

Tato konstrukce umožňuje vytváření jednoduchých podmínek (true/false), za kterých se dané volání provede či nikoliv, či specifikaci dalších vlastností

spojených s daným artefaktem. *Guard* se zapisuje do hranatých závorek k odpovídajícímu artefaktu (typicky před popis volání).
Jak bylo zmíněno, nejběžnější použití konstrukce *Guard* je určeno pro jednoduché pomínky; neumožňuje však větvení, jedná se vlastně o nejjednodušší variantu příkazu *if* bez *else*. Podmínka se zapisuje do hranatých závorek před popis volání.



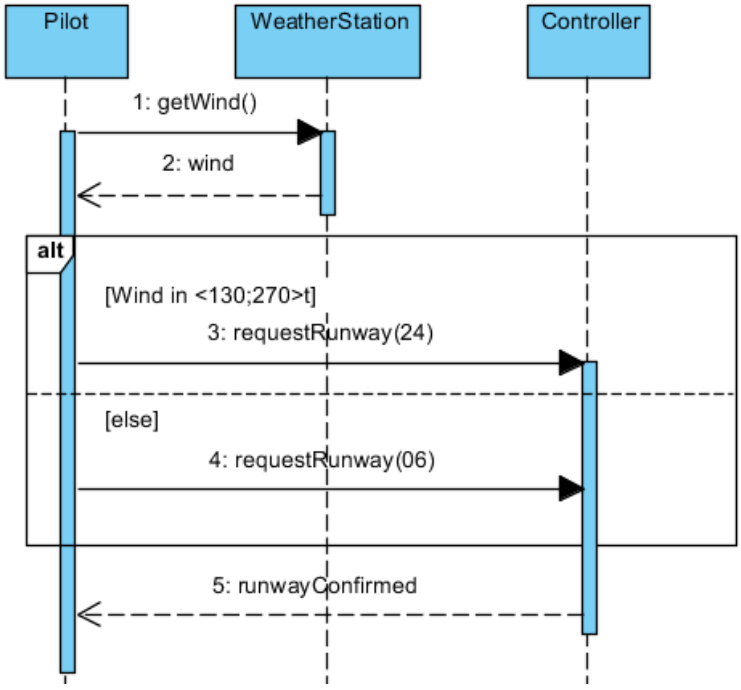
Další možnou variantou je jednoduchý zápis cyklu – například:

```
[for each airplane in airplanes]
```

10.8.4 Kombinované fragmenty – podmínky

Složitější variantou, kterou lze využít při modelování sekvenčního diagramu, je použití tzv. *kombinovaných fragmentů*. Kombinovaným fragmentem se rozumí jednoduchý **rámec**[33], který překryje životní čáry zainteresovaných objektů. Každý rámec má svůj odpovídající štítek, který říká, o jaký typ fragmentu se jedná.
Základním typem reprezentující výběr z několika možností je fragment *alternativa*. Jedná se o zápis, kdy může sekvenční diagram probíhat několika různými větvemi podle vyhodnocení podmínky. Jednotlivé části jsou alternativy od sebe odděleny přerušovanou čarou a stejně jako u prvku *Guard*, podmínky se zapisují do lomených závorek. Rámec *alternativy* obsahuje štítek **alt**.

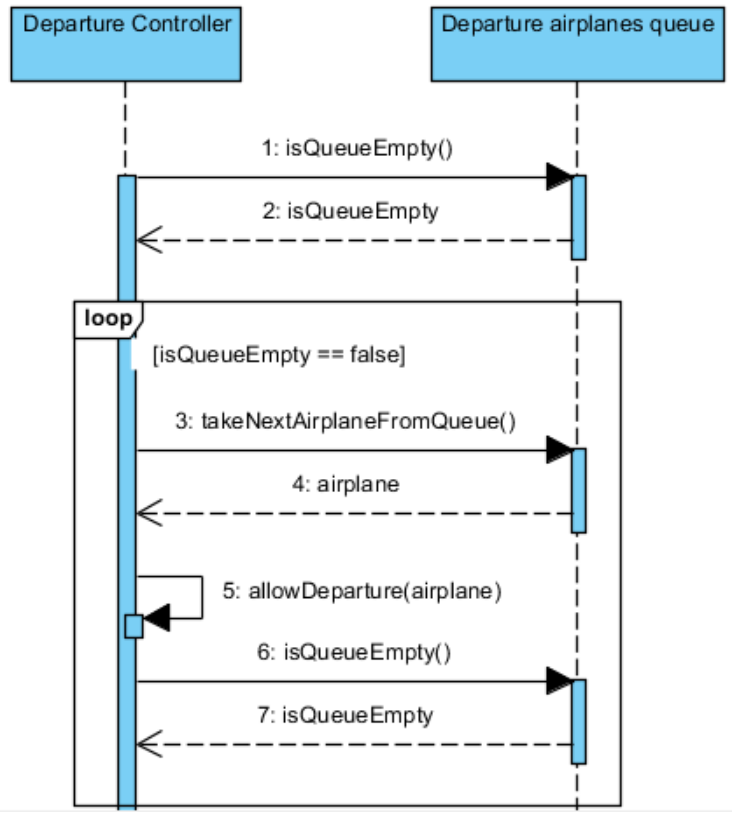
[33] Jedná se o stejný rámec, který byl představen při základním popisu každého diagramu. Tentokrát ale vlevo nahoře v rámci není uveden typ a název diagramu, ale jiný popis.



Ve výše uvedeném příkladu pilot požádá o ranvej 24, pokud jde vítr ze směru 130 – 270°, jinak požádá o ranvej 06.
Jednodušší obdobou alternativy je *option*, možnost – která nemá větev *else*. Jedná se o složitější *guard*, kdy do rámečku můžeme zapsat sadu různých interakcí (kdežto *guard* je vždy spojen pouze s jedním voláním). Rámec možnosti obsahuje štítek **opt**.

10.8.5 Kombinované fragmenty – cyklus

Další typickou návrhovou záležitostí sekvenčních diagramů jsou cykly. Využívá se stejného artefaktu jako u alternativ, tedy rámce, který zahrnuje všechny operace, které se mají provádět v cyklu. Rámec cyklu má štítek **loop**, navíc lze, stejně jako u rámce *alt*, do hranatých závorek specifikovat podmínku, která musí být splněna, aby se rámec provedl. Pokud podmínka není splněna, cyklus se opouští.



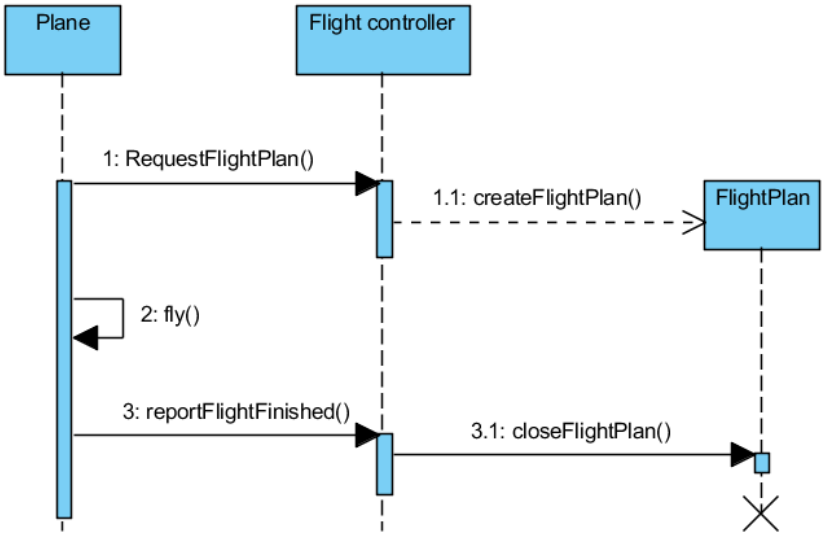
10.8.6 Kombinované fragmenty – ostatní

Kombinovaných fragmentů je více druhů a popis všech přesahuje rozsah této studijní opory. Běžně používané, které se mohou ještě hodit, jsou:

- Break (štítek *break*) – je poměrně podobný typu *alt* nebo *opt*. Obsahuje podmínku, která říká, kdy se do fragmentu vstoupí a operace v něm uvedené se provedou. Liší se ale tím, že pokud se příkazy ve fragmentu provedou, po ukončení fragmentu se již **neprovádí další příkazy** za tímto fragmentem (funguje tedy obdobně jako příkaz *break* v jazce Java).
- Parallel (štítek *par*) – reprezentuje jednoduchý fragment pro realizaci paralelních operací. Opět, jednotlivé paralelní bloky jsou od sebe odděleny vodorovnou přerušovanou čarou, jako varianty u fragmentu *alt*.

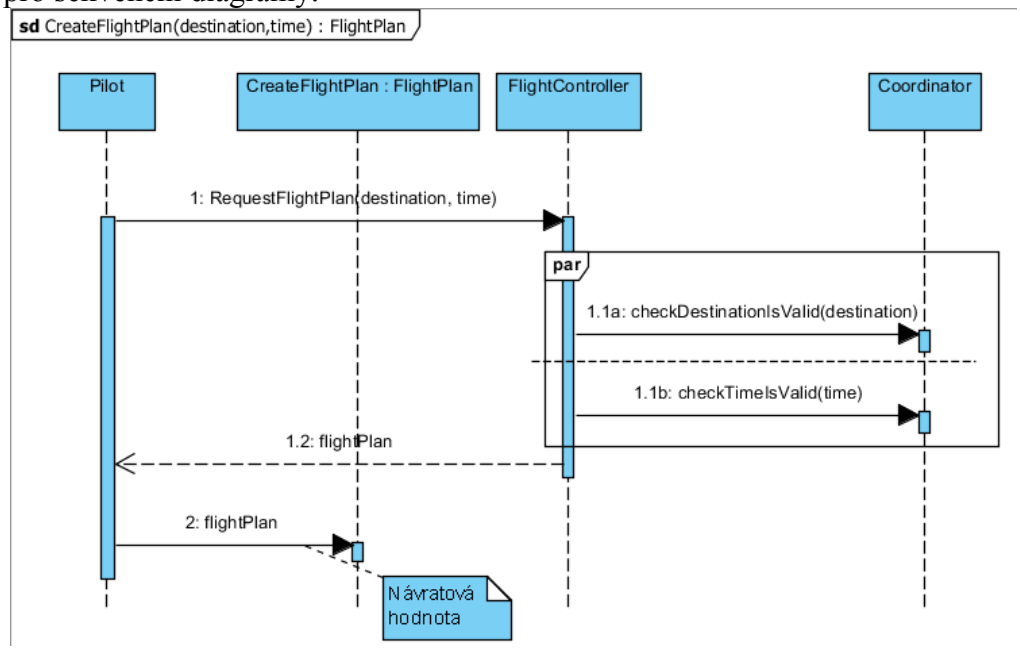
10.8.7 Vytváření a rušení objektů

Ne všechny objekty musí nutně v sekvenčním diagramu existovat od začátku. Typicky existují pouze byznys objekty nebo klíčové objekty daného sekvenčního diagramu. Voláním však libovolné objekty mohou jiné objekty vytvořit a posléze zrušit. Odpovídající syntax ukazuje následující obrázek.



10.8.8 Volání sekvenční diagramů, vstupní a výstupní parametry

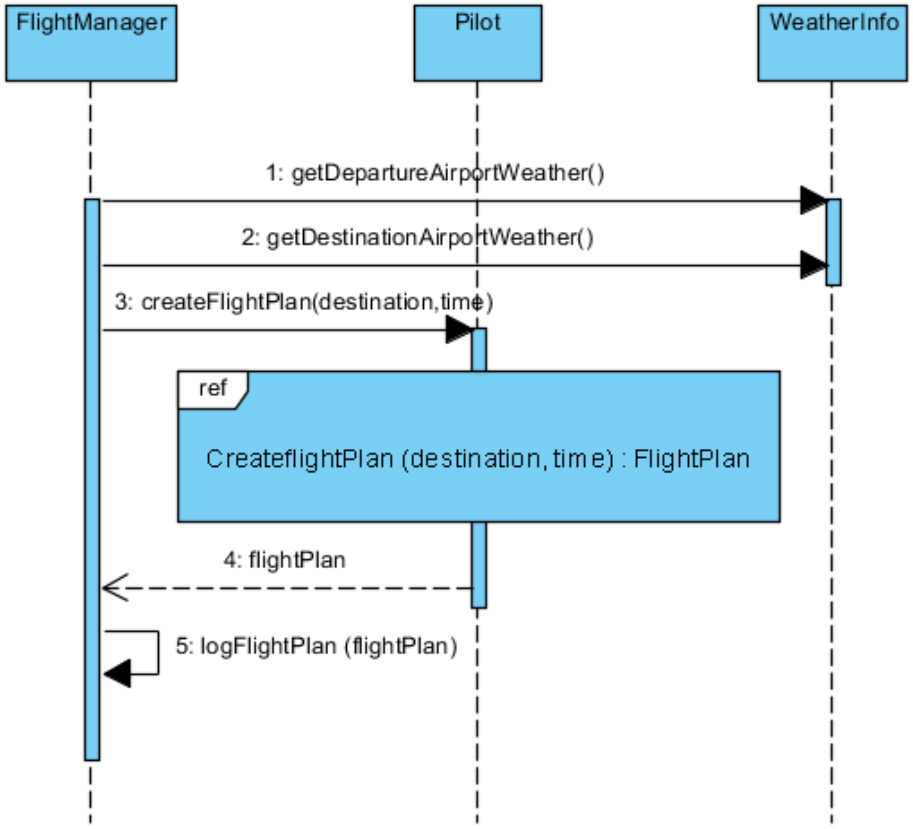
Sekvenční diagramy se již jen v lehce komplikovanějších případech mohou velmi rychle stát nepřehlednými. Proto se používají techniky, které umožňují zavolat jiný sekvenční diagram z aktuálně tvořeného. Celá technika funguje hodně podobně jako u programování a volání funkcí. Protože funkci při programování jsme schopni předat data ve formě parametrů – a stejně tak nám potom funkce může vrátit hodnotu, bude nejdřív představena stejná technika pro sekvenční diagramy.



Prvním bodem je název sekvenčního diagramu. Do závorek (podobně jako u funkcí) uvádíme seznam parametrů. Lze doplnit i datový typy, případně výchozí hodnoty, je-li třeba. Za dvojtečku uvádíme očekávaný typ, který sekvenční diagram vrací. Pokud neočekáváme návratovou hodnotu, blok opět neuvádíme.

Parametry lze zapisovat v sekvenčním diagramu jako vlastní objekty s životní čarou, vhodnější je ale přímo je používat při popiscích jednotlivých vytvářených volání (viz krok 1) – samozřejmě, **názvy parametrů uvedených v pojmenování sekvenčního diagramu musí odpovídat názvům uvedeným**

v popiscích. Jako návratová hodnota slouží opět zavedený objekt v rámci sekvenčního diagramu, který **je pojmenován stejně, jako sekvenční diagram.** V rámci sekvenčního diagramu mu potom požadovanou hodnotu ke vrácení vložíme (viz bod 2).
Nyní jsme tento sekvenční diagram schopni vyvolat a zjednodušit tak původní diagram.



K volání se opět využije kombinovaný fragment, se štítkem **ref**. Jako nápis do fragmentu se následně vloží text popisující, který jiný sekvenční diagram se má zavolat.

10.9 Diagram komponent

Diagram komponent představuje vztahy mezi jednotlivými komponentami systému. Byl dostatečně představen v předchozí studijní opoře. Blíže bude nyní představen pouze mechanismus rozhraní.

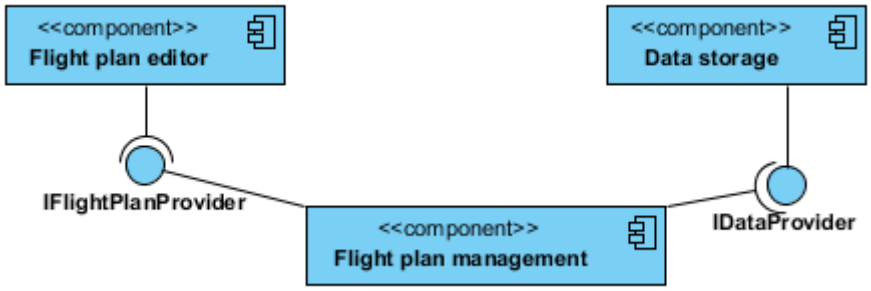
10.9.1 Rozhraní

Obecný mechanismus propojení komponent mezi sebou uvažuje, že komponenty se (ve smyslu závislostí) mezi sebou znají a vědí o sobě. Často se však dostaneme do problému cyklické závislosti, která se v programátorském prostředí při použití běžných principů řeší dost nesnadně Proto je pro doplnění vhodné zmínit mechanismus rozhraní pro demonstraci nezávislosti jednotlivých komponent na implementaci.
Technika rozhraní vychází z klasických programovacích principů, kdy potřebujeme, aby určitý blok mohl využívat nabízené funkcionality jiné komponenty, ale zároveň byl zcela odstíněn od konkrétní implementace. U rozhraní komponenta pouze říká, jaké služby (operace, funkcionality) nabízí, či jaké funkcionality vyžaduje. Komponenty v diagramu komponent tedy mohou

rozhraní jak nabízet (tj. implementovat), tak vyžadovat pro připojení. Komponenty představují rozhraní ve dvou přístupech:

- **Required** (tedy požadované) – značí se půlkruhem připojeným čarou k dané komponentě, typicky doplněný názvem rozhraní.
- **Provided** (tedy poskytované) – značí se kolečkem připojeným čarou k dané komponentě, typicky doplněný názvem rozhraní.

Pokud se dvě komponenty propojují přes rozhraní, zakreslí se kolečko dovnitř půlkruhu. Správné zakreslení zajistí korektní připojení správných typů rozhraní na sebe.

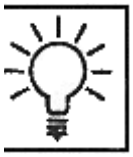


10.10 Shrnutí

(Zejména) pokročilé možnosti modelování však nemusejí nabízet všechny nástroje a v některých je třeba vystačit si s tím, co daný nástroj umí. Některé nástroje určitou problematiku mohou řešit pomocí vlastních značek či řešení. Důležité je, že **u diagramů ale není cílem precizní formalizace**, ale podání problematiky tak, aby i nově příchozí jednoznačně a pokud možno snadno pochopil, co se mu daný diagram snaží sdělit.

Kontrolní otázky:

1. Co je to stereotyp v UML a k čemu se využívá?
2. V jakém diagramu se mohou vyskytovat artefakty reprezentující třídy?
3. Čím se liší diagram komponent od diagramu tříd?



Úkoly k zamyšlení:

V kapitole byla uvedena problematika zaznačení programátorských konstrukcí do sekvenčních diagramů. Zdůvodněte, zda/kdy je to vhodné a kdy ne a ve které fázi životního cyklu projektu podle OpenUp se podobné artefakty v sekvenčních diagramech mohou začít vyskytovat?

Korespondenční úkol:
Nakreslete jednoduchý diagram případů užití, diagram tříd, sekvenční diagram a diagram komponent pro problematiku „Objednáváni pizzy online“.

Shrnutí obsahu kapitoly
V této kapitole byly přestaveny pokročilejší postupy a konstrukce při reprezentaci modelovaného informačního systému pomocí UML diagramů.

11 Vzory, anti-vzory

V této kapitole se dozvíte:

- Jaké známe druhy vzorů a jejich příklady.
- Co jsou to anti-vzory a k čemu slouží.
- Jaká je struktura anti-vzorů.

Po jejím prostudování byste měli být schopni:

- Aplikovat vzory v dané oblasti.
- Vyvarovat se anti-vzorům v dané oblasti.

Klíčová slova této kapitoly:

Návrhové vzory, anti-vzory.

Doba potřebná ke studiu: 4 hodiny

Průvodce studiem

Kapitola představuje problematiku návrhových vzorů a také problematiku anti-vzorů. Návrhové vzory jsou souborem znovupoužitelných vhodných praktik pro návrh software, anti-vzory pak popisují špatné řešení problémů a myšlenkový proces k nim vedoucí, což člověku pomáhá se těchto chyb vyvarovat. Nedílnou součástí anti-vzoru je pak také řešení.
Na studium této části si vyhraďte 4 hodiny.

S pojmem návrhové vzory se již pravděpodobně každý student setkal. Novým pojmem však budou vzory v procesní oblasti, vzory architektur a také tzv. anti-vzory (*anti-patterns*). Cílem tohoto textu není popisovat návrhové vzory, to je účelem mnoha jiných předmětů a knih (namátkou kurz OBJAP či výborná česká kniha od Rudolfa Pecinovského: *Návrhové vzory*. Computer Press. 2007). Naším cílem bude představit i další oblasti vzorů a také postupy či řešení, kterých se vyvarovat, tzv. anti-vzory

Co to tedy jsou vzory a anti-vzory a k čemu slouží? Vzory jsou abstraktní řešení definovaného problému oproštěné od technologie, prostředí či organizace. Jedná se o ověřená řešení v dané oblasti. Nejznámější jsou návrhové vzory, které nám říkají, jak řešit konkrétní problémy při návrhu software, např. jak vrstvit aplikaci, jak odstínit databázi či jinou technologii, jak vytvářet instance objektů a spoustu dalších řešení návrhových problémů. Dalšími možnými kategoriemi vzorů jsou vzory procesní, manažerské či vzory architektur. Přesto, že návrhové vzory by už neměly být pro čtenáře nové, začneme právě jimi, abychom měli výchozí bod pro další oblasti a základní pochopení, k čemu vlastně vzory používáme.

11.1 Návrhové vzory

Návrhový vzor je obecné znovupoužitelné řešení běžně se vyskytujících problémů při návrhu software. Návrhový vzor nepředstavuje konkrétní návrh přímo realizovatelný ve zvolené technologii. Jedná se o návod (šablonu, abstraktní popis) použitelný v mnoha rozdílných situacích, který popisuje, jak

řešit daný problém. Objektově orientované návrhové vzory prezentují řešení pomocí vztahů a interakcí mezi třídami či objekty, a to bez bližšího určení konkrétních implementací. Algoritmy nejsou považovány za návrhové vzory, jelikož řeší výpočetní problémy spíše než problémy návrhu. Ne všechny softwarové vzory jsou návrhové. Návrhové se zabývají přímo problémy na úrovni návrhu software, kdežto další druhy, např. vzory architektur popisují řešení, které má jiné zaměření, více o nich proto v následující kapitole.

Návrhové vzory nepochází ze softwarového inženýrství, jsou zcela běžné v každodenním životě. Nejvíce využívány jsou pravděpodobně ve stavitelství a v architektuře. Pro řešení určitých problémů je využíváno standardních řešení (sloupy jako podpora, oblouky pro přechod sloupů ke stropu, šikmé střechy, dveře pro vstup apod.), které jsou dodnes téměř neměnné. Dveře jako mechanismus pro vstup je ověřený vzor, ale sám o sobě nám ještě neříká žádné implementační detaily. Neřeší, zda použít dřevo, kov, sklo, plast, jaké panty, jakou velikost. Stejně je to s návrhovými vzory. Pouze říkají, co je vhodné udělat, ale jak záleží na technologii, kontextu a znalosti a zkušenostech programátora. Výhodou použití vzorů v jakékoliv oblasti, je zrychlení vývoje a lepší údržba a čitelnost kódu díky testovanému a prokázanému řešení.

Pro snadné použití vzorů je definována jejich následující struktura:

- Název – unikátní, popisné jméno vzoru, které napomáhá jednoduché identifikaci vzoru.
- Kontext – situace, ve které můžeme vzor uplatnit.
- Problém – který vzor řeší, včetně diskuse požadavků.
- Řešení – základní princip řešení.
- Struktura – detailní specifikace struktury vzoru a jeho aspektů.
- Dynamika – je popsána scénářem chování vzoru během jeho vykonávání.
- Implementace – hlavní body implementace vzoru.
- Varianty řešení.
- Známé použití vzoru.
- Výhody navrženého řešení.
- Slabá místa řešení.

Návrhové vzory dělíme do tří základních oblastí podle toho, jaké problémy řeší. Jedná se o:

- Vytvářecí (*Creational*) vzory – zabývají se vytvářením objektů, např.:
  - *Abstract Factory* – centralizované instancování továren.
  - *Factory Method* – centralizované instancování objektů konkrétních typů.
  - *Builder* – oddělení konstrukce složitého objektu od jeho reprezentace.
  - *Singleton* – omezení na vytvoření pouze jediné instance.
- Strukturální (*Structural*) vzory – zjednodušují návrh tím, že popisují jednodušší cestu, jak realizovat vazby mezi entitami, např.:
  - *Adapter* – upravuje rozhraní třídy do tvaru, který je předpokládán klientem.
  - *Facade* – vytváří zjednodušené rozhraní existujícího objektu, za účelem snazšího použití běžných úkolů.

    - *Proxy* – třída fungující jako rozhraní mezi dalšími elementy (sítí, souborem, velkým objektem v paměti apod.)
- Vzory chování (*Behavioral*) – popisují algoritmy nebo spolupráci objektů, např.:
    - *Observer*

Časté chyby při použití návrhových vzorů:

- Nepoužití žádného vzoru – výsledkem je moloch, špagetový kód, těžko udržovatelná aplikace, prostředí je náchylné k chybám.
- Použití vzoru na něco jiného (jiný kontext a účel) – opět těžko udržovatelné řešení, matoucí struktura kódu.
- Příliš mnoho vzorů – aplikace je převzorována a opět se stává nepřehlednou.
- Nepoužití názvu vzoru v dané třídě (třídách) implementující vzor (správně např. `class DBConnectionSingleton`) – výsledkem je nepochopení implementace, struktury kódu, obcházení předdefinovaných mechanismů.

Více informací a popisů vzorů viz klasické dílo v této oblasti od Gamma a spol.: *Návrh programů pomocí vzorů*. Grada. 2003, nebo Pecinovský: *Návrhové vzory*. Computer Press. 2007

11.2 Vzory architektur

Tato oblast vzorů má širší zaměření než návrhové vzory. Je zaměřena na poskytování řešení k problémům spojeným s architekturou, tj. základní model vrstvení a komunikace aplikace. Návrhové vzory řeší detailní problém v jedné vrstvě či na úrovni komunikace mezi nimi. Architektonický vzor vyjadřuje základní strukturu a organizační schéma softwarového systému, který se skládá ze subsystémů, jejich odpovědností a vzájemných vztahů. Jedná se tedy o koncept obsahující nezbytné elementy architektury, nejedná se o architekturu jako takovou. Bezpočet rozdílných architektur může totiž implementovat stejné vzory a tím pádem mít stejný charakter.

Velmi důležitým aspektem vzorů architektur je zahrnutí rozdílných nefunkčních požadavků (tzv. *qualitative attributes* – viz učební text Informační systémy 1). Například některé vzory reprezentují řešení výkonnostních problémů, jiné mohou být úspěšně použity v systémech s vysokou dostupností (*high-availability systems*). V raných fázích návrhu ***vybírá softwarový architekt takové architektonické vzory, které nejlépe naplní kýženou kvalitu systému***. Toto je základním pravidlem a prvním krokem návrhu systému! Návrh systému je řízen nefunkčními požadavky. Nyní by už mělo být čtenáři jasné, proč je neexistence nefunkčních požadavků takový problém a velmi často má za následek nevhodně zvolenou architekturu.

Vzory architektur je možné systémově rozdělit do několika kategorií (pohledů – *views*) daných jejich zaměřením. Některé jsou orientované na tok dat v systému (např. *pipes and filters*), další na pořadí výpočtů/zpracování dat (např. vrstvená či klient/server). Většina těchto typů architektur je podpořena

komerčními frameworky, konkrétně CORBA, EJB, DCOM (distribuovaná arch.), dále například Domino, mySAP platforma apod.

Základní vzory architektur podle [Pa1], [SEI1], [Pa2] jsou následující:

- ***Vrstvená*** – dalo by se říci, že se jedná o základní vzor architektur říkající, že pro redukci komplexnosti, rozdělení odpovědností (= lepší údržbu) a znovu použitelnost je třeba rozdělit aplikaci do relativně nezávislých vrstev, které spolu komunikují definovaných způsobem. Specifičtějším potomkem tohoto vzoru architektury je například vzor MVC.
- ***Klient/server*** – asi nejznámější vzor, u nějž je přesně viditelné, že se opravdu jedná pouze o vzor, ne o konkrétní architekturu. Vzor klient/server nedefinuje počet klientů, způsob komunikace mezi klientem a serverem či jejich rozhraní. Definuje pouze rozdělení do částí a jejich odpovědnosti. Tento vzor ale může být naplněm v různých technologiích s různými atributy.
- ***Distribuovaná*** – distribuované systémy potřebují jinou softwarovou architekturu, než nedistribuované, protože nedisponují sdílenou pamětí ani časem. Systém musí také odpovídat, i když je vzdálená komponenta mimo provoz, proto je nutné mít redundantní nezávislé objekty (vzor *Redundant Independent Object*), sdílené objekty či zdroje (vzor *Shared Object*) mezi klienty nebo řízení komunikace při volání služeb vzdálených objektů (vzor *Broker*).
- ***Pravidlová*** (*rule based*) – definuje strukturu aplikace, kde nejsou podnikové procesy svázány ve formě kódu, ale jsou definovány pomocí (podnikových) pravidel (tzv. *business rules*). Taková architektura pak ve fázi provozu umožňuje flexibilně měnit podporované procesy, přidávat nová pravidla či měnit stávající (příkladem mohou být BPM nástroje či SOA ve spojení s jazykem BPEL).
- ***Model-View-Controller*** – jedná se jak o návrhový vzor, tak o vzor architektury. Jako architektonický vzor rozděluje aplikaci do tří vrstev (které mnohdy běží na jiných HW strojích) s různými odpovědnostmi a definuje interakce mezi temito vrstvami. Model obsahuje klíčovou funkcionalitu a data, pohled (*View*) zobrazuje informace uživateli a kontrolór (*Controller*) řídí uživatelské vstupy. Vzor však již opět nespecifikuje, jaká technologie má být použita na kterou vrstvu či pro komunikaci mezi částmi. Model může být například reprezentován jako data v XML formě spolu s byznys pravidly sloužícími pro jejich transformaci či jako EJB vrstva aplikace.
- ***Pipes and filters*** – tento vzor poskytuje strukturu systému, který zpracovává proudy dat. Každý krok zpracování je zapouzdřen jako komponenta zvaná filtr, data jsou pak pomocí rour (*pipes*) předávána jednotlivým filtrům. Kombinací či přesunutím těchto filtrů jsme schopni vytvořit rodiny příbuzných systémů. Typickým příkladem jsou shelly operačních systémů.
- ***Orientovaná na služby*** – princip této architektury spočívá v zabalení funkčností do nezávislých celků zvaných služby, které spolu mohou být vzájemně kombinovány a jsou většinou dány do kontextu podnikového procesu a jako na takové modelovány na vyšší vrstvě abstrakce (např. pomocí jazyka BPEL – *Business Process Execution Language*). SOA je

opět vzor architektury, koncept, který může být implementován různými technologiemi (např. webové služby a technologie jako SOAP).
- Dalšími příklady jsou například *Shared repository*, *Reflection*, *Blackboard*, viz [Pa1], [Pa2].

Toto bylo stručné představení návrhových vzorů a vzorů architektury. Jelikož je náplní tohoto předmětu hlavně procesní oblast, tj. průchod životního cyklu projektu tvorby softwaru, představíme v následující kapitole procesní vzory, které nejsou tak známé jako návrhové vzory či vzory architektur.

11.3 Procesní vzory a anti-vzory

Vzory v procesní oblasti by pro čtenáře neměly být úplně nové, jelikož principy podle RUP se skládají ze dvou částí: vzor a anti-vzor. Vzor popisuje to, co funguje a jak to použít, antivzor pak, čeho se vyvarovat, co způsobuje problémy (více k principům RUP-OpenUP viz učební text Informační systémy 1). Vzory a anti-vzory představené v rámci RUP principů jsou však ještě docela konkrétní a nestrukturované, resp. neformální. Podívejme se, jak jinak lze ještě procesní anti-vzory definovat, aby odpovídaly struktuře, kterou známe z návrhových vzorů.

Následující text je upravenou českou citací článku [PG06]. Procesní anti-vzory popsány níže mají následující strukturu (velmi podobnou návrhovým vzorům):
- ***Název (name)*** – krátký popisný název anti-vzoru.
- ***Kontext (context)*** – popisuje problém, který se snažíme řešit.
- ***Hybné síly (forces)*** – popis hybných sil v organizaci, které nás přinutily přemýšlet o daném problému a donutily nás ho řešit.
- ***Řešení (solution)*** – podstata anti-vzoru, po uvážení hybných sil volíme zde popsaný postup. Tento postup provádíme proto, že vypadá logicky, zkušenost nám však říká, že taková řešení nakonec vedou k ještě větším problémům.
- ***Důsledky (consequences)*** – popisují negativní dopady vybraného řešení (rozuměj anti-vzoru).
- ***Alternativní řešení (alternate solution)*** – alternativní řešení, které bychom měli uvážit, abychom předešli negativním důsledkům.

Před vlastním uvedením těchto anti-vzorů je třeba říci, že se nejedná o empiricky ověřené a IT komunitou „schválené“ vzory, ale nejen dle autorovi a mé zkušenosti jsou toto až příliš časté chyby. Z akademického pohledu je třeba říci že se nejedná o výstup výzkumu, ale spíše o počáteční vstupy resp. hypotézy, které je nutno dále prozkoumat.

Název	**Ber daný lék naráz**
Kontext:	Proces, stejně jako léky je dobrý pomocník. Proces pomáhá vnést pořádek do organizace a přispívá k plynulému chodu věcí. V určitém okamžiku organizace usoudí a rozhodne, že je třeba vylepšit proces vývoje software.
Hybné síly:	Proces je důležitý, je třeba abychom věděli, co kdy dělat, předávali znalost dalším týmům a také prošli audity. Musíme být tak efektivní, jak je jen možné, proto předáváme znalost dále ve formě procesních popisů. Na trhu existuje mnoho knih či konzultantů, kteří nám pomohou s přechodem od našeho ad hoc

	procesu k efektivnějším postupům. Managementu se líbí myšlenka zlepšení procesu a dal nám plnou podporu.
Řešení:	Rozhodli jsme se implementovat celý životní cyklus vývoje software napříč organizací. Připravili jsme tréninkové materiály, najali jsme nejlepší konzultanty a připravili datum, kdy představíme náš proces. Odteď bude každý postupovat konzistentně, předvídatelně a tyto kroky mohou být měřeny a neustále vylepšovány. Proces nám také pomuže vyvarovat se chybám.
Důsledky:	Bohužel, tento přístup nezafungoval a věci se nezlepšily, spíše zhoršily. Změna vyžaduje čas, aby si na ni lidé zvykli, aby nezpůsobila zásadní problémy, snížení efektivnosti či demotivaci lidí. Změny je třeba pojmout po malých krocích. V případě zásadní změny tráví lidé čas pouze hledáním informací a studiem, jak že kterou činnost mají provést nyní a nezbývá jim žádný čas na vlastní práci, případně se vzdají následování nového procesu a vykonávají věci po staru.
Alternativy:	Dávkuj nové prvky procesu po malých krocích. Identifikuj klíčové oblasti, kde by zlepšení mohlo přinést největší hodnotu. Implementuj změny v této oblasti, ohodnoť výsledky a poté se připrav na další malý krok/změnu v další oblasti. Pokud sesbíráš dobrá data vypovídající o tom, jak změna pomohla, každý tomuto bude rozumět a bude ochotný k další změně.

Tento anti-vzor je bohužel velmi často následován. Často zahazujeme vše, i fungující inovativní postupy (aniž to mnohdy tušíme), jelikož nejsou součástí nového procesu.

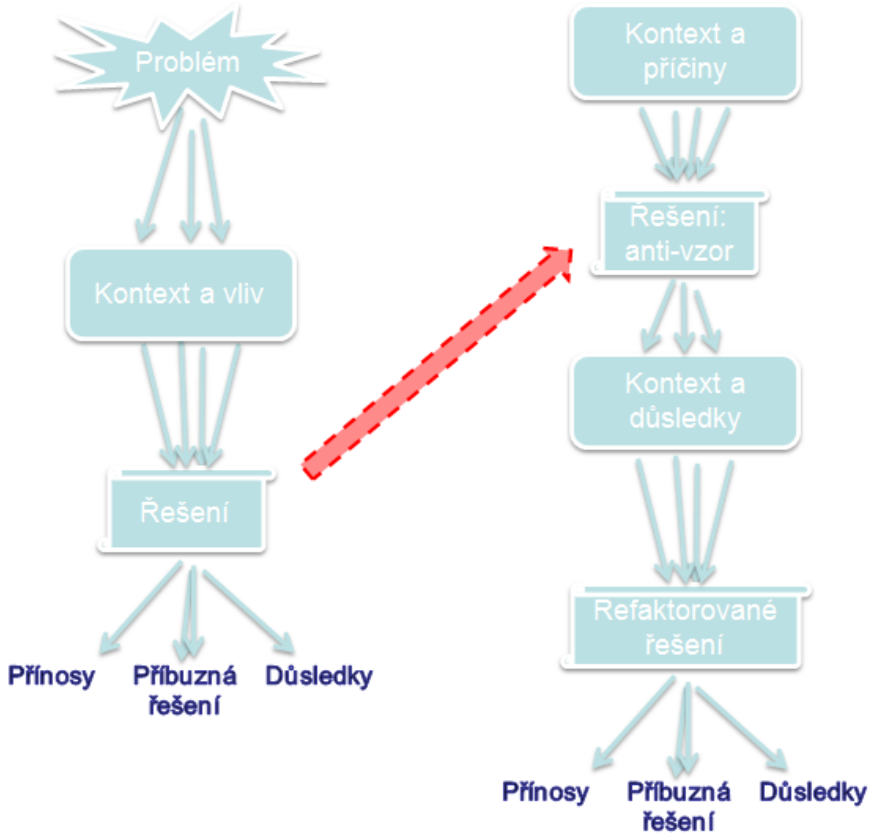
Název:	**Malé inkrementy, krátké iterace**
Kontext:	Rozhodli jste se rozjet projekt zlepšování procesu až po projektu definice softwarového procesu: iterativně a inkrementálně. Softwarové projekty se zdají být úspěšné pokud vytváříme malé inkrementy v krátkých iteracích.
Hybné síly:	Potřebujete zlepšit procesy a jste pod tlakem provést tuto změnu tak rychle, jak je jen možné. Myslíte si, že malé inkrementy umožní okamžité zlepšení. Jakmile dokončíte nasazení jedné změny, začnete s druhou. Lidé se v daný okamžik učí pouze jednu věc, takže vypadá logicky, že tímto způsobem můžete zkrátit celkovou dobu trvání změny procesu a sklízet přínosy změn okamžitě po její implementaci = efekt je kumulativní.
Řešení:	Vytvoříte tým, který bude změny nasazovat do reality, vytvoříte školící materiály a uspořádáte školení pro každou změnu a naplánujete je v těsném sledu, tak, že jedna změna metody (či metoda nová) následuje druhou pouze s malou odmlkou.
Důsledky:	Tento přístup vede ke zmatku. Jen to, že vám někdo něco řekl, vyškolil vás, neznamená, že dané věci rozumíte a můžete ji používat. Lidem trvá různou dobu, než alespoň trochu pochopí nové postupy, metody. Zřetězení nových změn rychle za sebou a jejich podávání lidem ve skutečnosti prodlouží dobu nezbytnou k jejich pojmutí a pochopení a může vést k celkovému neúspěchu zlepšení vývojového procesu.
Alternativy:	Identifikujte malou množinu změn, které se týkají specifické oblasti organizace či procesu. Příklad: vyškolte členy týmu v použití use case pro specifikaci a řízení požadavků, systémový návrh a testování. Nechte tým používat nové techniky v průběhu celého projektu, resp. releasu. Pak představte další skupinu změn a metod. Věci chtějí svůj čas. Pokud nenecháte tým, aby si zvykl na nové metody, začal je používat v kontextu jejich projektu, máte malou šanci, že se stanou úspěšnými a používanými. Toto je přesně oblast, kde platí práce kvapná málo platná.

Toto byl stručný příklad dvou anti-vzorů v procesní oblasti, který názorně ukazuje (hlavně druhý z nich), jak mohou být dobré myšlenky chybně aplikovány. Pojďme se nyní zabývat anti-vzory v oblasti vývoje software.

11.4 Anti-vzory

V předchozí kapitole jsme si představili procesní anti-vzory. Nyní se jimi budeme zabývat na úrovni software: architektury a kódu. V dané oblasti existuje katalog různých druhů vzorů, který kromě jiného také rozšiřuje definici toho, co to je anti-vzor. Tato definice nám tedy říká, že anti-vzor [AP1]:

- Je mapování obecného řešení na specifickou třídu řešení – redukce nevhodného použití anti-vzoru v jiném kontextu.
- Reprezentuje zkušenost: opakující se chyby a jejich řešení (včetně kořenových příčin).
- Poskytuje slovník pro oblast identifikace problému a diskusi řešení – lepší komunikace.
- Poskytuje holistické řešení konfliktů (uvažovány všechny úrovně – organizační, řídící, vývojářská, …).
- Poskytuje seznam běžných nástrah, úskalí v SW průmyslu.



Obr. 11-1: Koncept vzorů a anti-vzorů (struktura a smysl jednotlivých částí), zdroj: [AP1]

Obrázek ukazuje koncept vzorů a anti-vzorů. Struktura návrhového vzoru je problém a jeho řešení. Návrhové vzory předpokládají řešení/projekty na zelené louce a jsou těžko čitelné právě díky své bastraktní povaze. Anti-vzory mají rozdílnou strukturu:

- První řešení – problematické, běžně se vyskytující řešení mající negativní důsledky, může se jednat i o aplikaci vzoru.

- Druhé řešení – refaktorované řešení mající přínosnou formu.

Anti-vzory počítají spíše s existujícími projekty a jejich změnou (což je v dnešní době realita spíše než projekty na zelené louce). Poskytují popis současného nevhodného řešení, kde se může každý najít a poskytují také lepší, refaktorované řešení, které vede k lepším výsledkům s minimem dopadů.

Podle [AP1] můžeme anti-vzory rozdělit do tří kategorií:

- ***Vývojářské anti-vzory*** – jsou určeny pro programátory řešící programátorský problém, jsou tedy rozšířením návrhových vzorů.
- ***Anti-vzory architektur*** – systémové problémy ve struktuře aplikací a systémů, jejich důsledky a řešení, jedná se tedy o rozšíření vzorů architektury.
- ***Anti-vzory řízení*** – problémy způsobené organizací, způsobem řízení projektů a jejich řešení (organizace ovlivňuje lidi, kteří tvoří výsledné řešení/aplikaci).
- Uvedli jsme si také čtvrtou kategorii – ***procesní anti-vzory***.

Stejně jako návrhové vzory, mají i anti-vzory svoji strukturu. Tato struktura může být vyjádřena jako miniverze či úplná verze.

Miniverze:

Název: Jak by měl být anti-vzor nazýván
Problém anti-vzoru: co je opakujícím se problémem s negativními důsledky?
Refaktorované řešení: jak se můžeme vyhnout tomuto problému či jej minimalizovat, refaktorovat?

Tento formát může obsahovat obrázek či vtip (neformálně popsaný účel existence).

Úplná verze:

Název: název, cíleně pejorativní, hanlivý
Znám také jako (alias): další jména
Nejběžnější oblast vzoru: kam zapadá anti-vzor (oblast vzoru a řešení – mikro-architektura, framework, aplikace, …)
Název řešení: identifikace vzoru řešení
Typ řešení: typ akce řešení (software, technologie, proces, role)
Příčiny: kategorie příčin (spěch, apatie, těžkopádnost, omezenost, …)
Nevyvážené síly: co bylo opomenuto (správa funkcionalit, výkonnosti, IT zdrojů, …)
Symptomy a důsledky: příznaky a důsledky
Refaktorované řešení: popis refaktorovaného řešení
Příklad: ukázková demonstrace řešení

11.4.1 Vývojářské anti-vzory

V této podkapitole si stručně představíme základní a běžně následované vývojářské anti-vzory podle [AP1] a jejich řešení. Nejběžnější anti-vzory v této kategorii jsou následující:

The Blob (kaňka, hrudka): procedurální návrh vedoucí k jednomu "mega" objektu s mnoha odpovědnostmi, zbytek objektů spravuje pouze data či vykonává velmi jednoduché procesy.
Řešení: refaktorovat třídy – rovnoměrná distribuce odpovědností na objekty společně s izolováním dopadu budoucích změn (nutnost změny pouze na jednom místě).

Lava Flow: nepoužívaný kód a zapomenuté návrhové informace po mnoha změnách návrhu – nikdo neví, k čemu tyto části slouží, zda něco dělají. Vývojáři se na něj bojí vůbec sahat, aby nerozbili fungující funkčnosti.
Řešení: Konfigurační proces eliminující mrtvý kód a umožňující refaktoring návrhu za účelem dosažení vyšší kvality (přehlednější a čitelný kód).

Ambigous Viewpoints (nejasné pohledy): spousta pohledů bez upřesnění, o jaký pohled se jedná, často používány modely s nejmenší přidanou hodnotou (díky tomu nelze oddělit implementační detaily od rozhraní – nelze využít základní benefit OOP!).
Řešení: vysvětlení, který pohled je použit (byznys, specifikační, implementační).

Spaghetti Code: náhodná struktura software, která dělá řešení z pohledu kódu těžce udržitelné a neoptimalizovatelné.
Řešení: častý refaktoring jednotlivých částí, prováděný krok po kroku (ne všechno naráz!) s podporou unit testů, které jsou pojistkou při refaktoringu.

Úplný anti-vzor Špagety kód:

Název: Špagety kód
Oblast vzoru: aplikace
Název refaktorovaného řešení: refaktoring software, vyčištění kódu
Typ řešení: software
Příčiny: neznalost, neinformovanost, lenost
Nevyvážené síly: řízení, resp. boj se složitostí a změnami
Symptomy a důsledky:

- prozkoumáme-li takový kód, zjistíme, že pouze některé objekty či metody lze znovu použít (ne celý balíček, modul),
- tok provádění je řízen implementací, ne klientem objektu,
- mezi objekty jednoho balíčku, modulu, jsou minimální vazby,
- množství metod nemá žádné parametry,
- základní výhody objektově orientované implementace jsou ztraceny: polymorfismus (návrh pomocí rozhraní), dědičnost,
- údržba se stává neúměrně drahou, levněji vyjde aplikaci znovu napsat, než tu současnou udržovat.

Typické příčiny:

- nezkušenost s objektovým návrhem,
- chybějící mentoring,
- neefektivní revize kódu,
- žádný návrh před vlastní implementací (ad hoc přístup),
- často výsledek vývojářů pracujících v izolaci.

Refaktorované řešení:

1. Použití refaktoringu, krok po kroku přepsat aplikaci: změna struktury a čitelnosti kódu beze změny chování aplikace.
2. Unit testy a systémové testy by měly být základním kontrolním mechanismem, pojistkou, že se chování aplikace nemění a že přepisem nevnášíme nové chyby a defekty.

Řešení v případě nové aplikace: prevence (vytvořte nejdříve doménový model pro pochopení požadavků, pak analytický, který pomůže identifikovat vrstvy/subsystémy aplikace, následuje hrubý návrh, definice pravidel pro psaní kódu, aplikace potřebných architekturálních či návrhových vzorů a pak teprve začněte psát kód).
Řešení v případě existující aplikace: pokud přijde nový požadavek či je nutné opravit chybu, před psaním nového kódu do existujícího nejdříve tento upravte, refaktorujte do lépe strukturované, čitelnější podoby (vyšší abstrakce – argumenty jako kolekce, nový balíček; možnost znovupoužití v jiných částech kódu; obstranění nepoužívaného kódu).
Příklad: podívejte se na jakýkoliv váš kód napsaný v průběhu studia či prvních let programování a srovnejte ho s nějakým Open Source kódem.

Golden Hammer*:* aplikace pouze jedné či několika známých technologií či konceptů na všechny softwarové problémy, které řešíme.
Řešení: rozšíření znalosti vývojářů i na alternativní technologie a přístupy (vzdělávání, tréninky, případové studie – case study).

Další vývojářské anti-vzory:

- *Continuous Obsolescence* – problémy s releasy (verzemi) díky rychlé změně technologií.
- *Functional Decomposition* – strukturální způsob programování v OOP.
- *Poltergeists* – třídy s velmi omezenou rolí, často startují procesy pro jiné objekty.
- *Boat Anchor* – neužitečný kód či hardware, nevykonávající žádnou činnost či nemající žádný smysl. Většinou důsledek akvizic či nákupů bez vize.

11.4.2 Anti-vzory architektury

V této části si stručně představíme základní a běžně následované anti-vzory architektury podle [AP1] a jejich řešení. Anti-vzory architektury se zaměřují na úroveň systému, resp. podniku. Jak již víme z předchozího textu a z Informačních systémů 1, je architektura kritickým faktorem úspěchu při vývoji software. Architektura nám poskytuje pohled na celý systém. K tomu, abychom mohli bojovat se složitostí celého řešení, využíváme většinou oddělené pohledy na různé aspekty systému. Architektura softwaru se zaměřuje na tři aspekty návrhu:

- funkční rozdělení systému do modulů,
- identifikace rozhraní mezi systémy,
- výběr a vlastnosti technologie použité pro implementaci spojení pomocí rozhraní mezi moduly.

Nejběžnější anti-vzory v této kategorii jsou tedy následující:
Stovepipe Enterprise: takový systém je charakteristický strukturou, která brání změnám, není možné znovu použít jeho části, složitá interoperabilita (spolupráce mezi částmi řešení či jinými systémy).
Řešení: abstrakce subsystémů, komponent; plánování a koordinace systémů v podniku (podniková architektura).

Swiss Army Knife: komplexní rozhraní třídy vytvořené autorem za účelem všemožného použití třídy (veškeré možné případy); naplnění veškerých potřeb.
Řešení: definice jasného účelu komponenty; abstrakce rozhraní; vytvoření profilů pro různé použití dané implementace.

Reinvent the Wheel: všudypřítomný nedostatek předávání technologické znalosti mezi projekty vede ke značnému znovu vynalézání.
Řešení: znalost návrhu zakódovaná ve formě znovupoužitelných komponent (naše vlastní, komponenty třetích stran, Open Source) redukuje náklad, rizika a čas nutný k dodání řešení na trh. Tento přístup také zvyšuje kvalitu dodávaného řešení a umožňuje znovu použít fungující řešení.

Název anti-vzoru: Znovu vynalézání kola
Známý také jako: návrh ve vakuu, návrh na zelené louce
Oblast vzoru: systém
Název refaktorovaného řešení: prozkoumání architektury
Typ řešení: proces
Příčiny: neznalost, neinformovanost, pýcha, hrdost
Nevyvážené síly: řízení změn, předávání technologické znalosti
Symptomy a důsledky:

- uzavřená architektura navržená pouze pro daný systém bez možnosti (úvahy) znovu použití či komunikace s dalšími systémy,
- nedostatečně vyvinuté a nestabilní požadavky a architektura,
- nedostatečná podpora řízení změn a interoperability,
- chabé řízení rizik a nákladů způsobující překročení rozpočtu a dohodnutého času projektu,
- neschopnost dodat požadované funkčnosti koncovým uživatelům.

Typické příčiny:

- žádná komunikace a chybějící transfer technologické znalosti mezi vývojovými projekty,
- absence návrhových postupů týkajících se architektury (doménové inženýrství, zkoumání architektur),
- předpoklad vývoje systému na zelené louce.

Refaktorované řešení: zkoumání existujících architektur (starší systémy, komerční produkty, standardy, návrhové vzory nám poskytnou možná řešení našich problémů). Toto zkoumání je možné spíše na úrovni podniku než na globální úrovni z důvodu přístupu k informacím. Jedná se o přístup zdola-nahoru zahrnující znalost návrhu z existujících implementací. Před vlastním zkoumáním je třeba identifikovat (rozhovory, konference, case study) skupinu relevantních technologií pro daný návrhový problém.

1. krok – modelování systému za použití dané technologie za účelem identifikace rozhraní.
2. krok – vytvoření specifikace rozhraní (generalizace modelovaného systému).
3. krok – úprava a pilování návrhu (podle názoru architektů, návrhářů, výsledků revizí).

Příklad: je třeba uvážit kompromis mezi požadovanou funkčností a možností znovu použít existující řešení, př.: tvorba grafického editoru na zelené louce vs. využití Eclipse a GEF (*Graphical Editing Framework*).

Další anti-vzory architektury:

- *Autogenerated Stovepipes* – použití stejného návrhu a rozhraní při migraci současného řešení na distribuovanou architekturu. Tento přístup způsobí mnoho problémů nejen s bezpečností dat či výkonností systému.
- *Vendor Lock-in* – uzamčení u jednoho dodavatele a jeho proprietární architektury, řešením je použití nezávislé mezivrstvy (*middleware*), např. SOA vrstva, tzv. ESB architktura založená na webových službách.

- *Wolf Ticket* – produkt deklarující otevřenost a standardy, ve skutečnosti je však dodáván s proprietárními rozhraními, které se velmi liší od standardů v dané oblasti.
- *Warm Bodies* – rozdílná kvalita a produktivita programátorů, zkušení programátoři jsou nutností každého projektu, učitelé a tahouni juniorských programátorů; někteří programátoři jsou na projektu bohužel jen do počtu; správný poměr senior-junior by měl být přibližně 1 senior : 2-4 juniorům.

11.4.3 Anti-vzory řízení

V této části představíme základní a běžně následované anti-vzory řízení softwarových projektů podle [AP1] a jejich řešení. Nejběžnější anti-vzory v této kategorii jsou následující:

Death by planning: nadměrné a příliš detailní plánování v úvodu projektu, navíc často neodpovídající realitě, zaměřeno na náklady spíše než na doručení řešení včas, bez ohledu na spoustu neznámých faktů, rizik, odsunutí vlastní implementace.
Řešení: inkrementální plánování, zaměření se na doručitelné artefakty (komponenty, produkt) s častou demonstrací zákazníkovi, zahrnutí ponaučení.

Název anti-vzoru: Mrtvý plánováním
Znám také jako: detailistické plánování
Oblast vzoru: podnik
Název refaktorovaného řešení: rozumné plánování
Typ řešení: proces
Příčiny: skoupost, neznalost, neinformovanost, spěch
Nevyvážené síly: boj s komplexností
Symptomy a důsledky:
- neschopnost plánovat na praktické, střízlivé úrovni,
- zaměření se na náklady spíše než na dodání (dodávání) aplikace,
- chamtivá snaha zavázat se k jakémukoliv detailu až poté, co je zajištěné financování,
- strávení více času plánováním, detailním sledováním postupu prací a přeplánováním, než na samotném tvoření a doručování software,
- neustálé zpoždění dodávek a možný případný nezdar projektu.

Typické příčiny:
- neexistující plán dodávek komponent a jejich data,
- neznalost základních principů projektového řízení,
- horlivé úvodní plánování za účelem dosažení absolutní kontroly nad vývojem,
- plánování jako hlavní projektová aktivita.

Refaktorované řešení: zachytit v projektovém plánu hlavně dodávky (přírůstky aplikace a dokumentace) a jejich data, ty by měly být identifikovány na dvou úrovních:
- produkt – artefakty prodávané zákazníkovi, využívané byznysem.
- komponenty – technologické artefakty nutné pro podporu služby.

Příklad: realita versus příklady projektových plánů v ROPR.

Analysis Paralysis: snaha o perfekcionalismus a kompletnost v analytické fázi vede k zabrzdění projektu a nadměrné odpadovosti modelů/specifikací.
Řešení: inkrementální a iterativní vývojový proces, který odkládá detailní analýzu až na místo, kde je třeba ji provést (na konkrétní iteraci).

The Freud: Osobní konflikty mezi manažery mohou významně ovlivnit pracovní prostředí. Jejich podřízení trpí díky nesouhlasu jednoho z nich či díky (často protichůdným) rozhodnutím, která přichází z různých stran.
Řešení: neformální sezení sloužící k poznání lidí a vyjasnění rozdílných názorů (kontextu stojícím za tímto rozhodnutím), MBTI jako nástroj pochopení potřeb a chování druhých.

Další manažerské anti-vzory:

- *Viewgraph Engineering* – nucení programátorů produkovat grafy, modely a dokumenty místo psaní kódu. Jak víme, že je daný model správný, když není ověřen kódem?
- *E-mail is Dangerous* – e-mail je důležitým prostředkem, ale nenahrazuje komunikaci; zaslaný e-mail nelze v žádném případě brát jako oznámení.
- *Throw it over the Wall* – objektově orientovaný přístup, návrhové vzory, plány, prostě všechny tyto flexibilní návody jsou brány manažery či vývojáři doslovně.

11.5 Prozřetelný pan Brooks

V předchozím textu jsme diskutovali vzory a anti-vzory z různých oblastí vývoje, provozu, údržby a řízení IS. Jak jsme si ukázali, je množství anti-vzorů viditelných zvláště v oblasti řízení projektů. V softwarovém inženýrství jsou známé také anti-vzory popsané Frederikem Brooksem v jeho knize novel [BF75] již v roce 1975. V roce 1995 vyšla tato útlá, ale veleúspěšná kniha, znovu i s komentářem autora. Bohužel, spousta popsaných antivzorů je následována i tak dlouhé době a dokonce je brána jako standardní praktikum v IT. Tím nejznámějším a také nejvíce následovaným je anti-vzor popaný v novele ***Mythical Man-Month***. Novela hovoří o přibrání nových lidí do vývojového týmu, pokud se projekt zpožďuje. Selský rozum říká, že pokud jsme ve skluzu, je dobré přizvat někoho na pomoc. Bohužel právě selský rozum v tomto případě selhává. Tento typický anti-vzor, tzv. ***Brooksův zákon*** (*Brooks' law*) způsobí naopak ještě další problémy:

- Snížení produktivity stávajícího týmu – stávající lidé v týmu se musejí věnovat nově příchozím, vysvětlit jim problémovou doménu, use casy a scénáře, proces vývoje, používané prostředí, architekturu produktu, mechanismy architektury a pravidla psaní kódu.
- Snížená produktivita týmu generuje další zpoždění projektu.
- Nutnost častější komunikace díky více lidem v týmu: komunikace s více lidmi – více komunikačních kanálů, které narůstají geometrickou řadou.
- Všechny tyto aspekty způsobují demotivaci stávajícího týmu.

Brooks dále zmiňuje, že mnohem důležitějšími faktory úspěšného projektu jsou lidé a jejich zkušenosti a znalosti, jejich organizace, pravomoce a řízení na rozdíl od nástrojů či technických postupů. Realita je taková, že podniková architektura a kultura je tvořena, dále rozvíjena a také následována lidmi. Brooks uvádí, že veškeré půtky týkající se správnosti a použití toho kterého postupu, notace či paradigmatu jsou bezvýznamné, pokud nemá podnik živou (= lidmi následovanou) strategii efektivní spolupráce. Ve větších firmách můžeme často vidět jednotlivé jednotky soupeřit místo toho, aby spolupracovali a přinášeli hodnotu zákazníkovi. Toto je důsledek nevhodně nastavených metrik (na osobní úrovni místo týmové), jejich odpojení od cílů organizace (či přímo jejich protichůdnost) a také právě důsledek neexistence či nefunkčnosti podnikové architektury.

Podobných pozorování pana Brookse je v této knize novel popsáno několik. Brooks pracoval v IBM na vývoji několika zásadních počítačů a systémů a jako vedoucí různých projektů se snažil s danými problémy bojovat. Pokud čtenáře problematika zaujala, pak doporučíme buď původní soubor novel [BF75] nebo reedici z roku 1995.

Z objektivního hlediska je třeba zmínit, že některé ze zmíněných problémů můžeme snížit či odstranit právě agilním myšlením a použitím agilních praktik na projektech. Agilní praktiky jsou zaměřeny na lidi a jejich spolupráci, sdílení znalostí a také stavu projektu a brzké odhalování a řešení problémů. Jako další důkaz z mnoha uvedeme kromě naší zkušenosti konkrétní zkušenost pracovníků Menlo Innovations publikovanou na konferenci XP 2007 [Op07].

Kontrolní otázky:

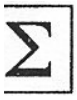
1. Co je to anti-vzor a čím se liší od vzoru?
2. Jaký je rozdíl mezi návrhovým vzorem a vzorem architektury?
3. Co je hlavním přínosem anti-vzorů?
4. Co je to Brooksův zákon?

Úkoly k zamyšlení:

Vybavte si, či znovu projděte, všechny anti-vzory ze všech oblastí, které byly představeny. Vidíte některý z nich ve vaší denní práci či při plnění vašich úkolů? Zamyslete se nad tím, jak jej nyní odstranit. Jejich identifikace je první důležitý krok. Jaké budou další?

Korespondenční úkol:

V úkolu k zamyšlení jste měli za úkol identifikovat, zda náhodou nenásledujete některý z představených anti-vzorů. Nyní se zamyslete obecně. Zkuste popsat postup, kterým byste anti-vzor identifikovali a jak jej v praxi odstranili. Vyberte si pouze jednu oblast (vývojářské, procesní, řídící, ...), jelikož pro různé oblasti bude tento postup pravděpodobně odlišný.



Shrnutí obsahu kapitoly

V této kapitole jsme představili problematiku vzorů v jiných oblastech, než je známé a také tzv. a anti-vzory. Konkrétně se jednalo o procesní, architektonické, vývojářské a řídící vzory i anti-vzory. Základním přínosem anti-vzoru je zachycený myšlenkový proces vedoucí k chybnému řešení.

Logickým rozšířením je tedy vhodnější řešení, které je také součástí popisu anti-vzoru. Kapitola byla hojně doplněna konkrétními příklady.

12 Měkké aspekty vývoje

V této kapitole se dozvíte:

- Co je tým a jak se liší od skupiny?
- Jaké techniky lze použít pro motivaci týmu?
- Jak je pro komunikaci důležitá znalost kontextu.
- Co je to MBTI?
- Co je a k čemu slouží koučink a mentorink?

Po jejím prostudování byste měli být schopni:

- Definovat tým a popsat fáze formování týmu.
- Porozumět odlišnosti kolegů a využít znalosti typologie při práci v týmu.
- Identifikovat svůj osobnostní typ podle MBTI.

Klíčová slova této kapitoly:

Tým, lidské aspekty vývoje software, typologie, MBTI, koučink, mentorink.

Doba potřebná ke studiu: 3 hodiny

Průvodce studiem

Kapitola se týká oblasti měkkých schopností a dovedností, který by měl každý člen týmu mít či o nich přinejmenším vědět a snažit se je využívat a zdokonalovat při každodenní mezilidské komunikaci. Dále kapitola objasňuje také pojmy mentoring a koučing.
Na studium této části si vyhraďte 3 hodiny.

Jelikož je software tvořen a dodáván lidmi, je nutné se zmínit o práci v týmu a mezilidské komunikaci. Právě tyto dva aspekty zásadně ovlivňují kvalitu doručeného produktu, výkonnost organizace a spokojenost týmu a jedinců uvnitř týmu. Co uvažujeme a zahrnujeme pod lidské faktory a aspekty? Jedná se především o pochopení osobních a týmových potřeb, principy vnitřní motivace a vnější inspirace (hygienických faktorů), ale také reflexy a reakce mozku a nervového systému, které jsou nám lidem přirozené a vyvíjely se po desítky tisíc let. Průmyslová revoluce a znalostní ekonomika toto přirozené chování nezmění během několika málo generací.

Měkké aspekty, jako je vůdcovství, motivace či emoční inteligence EQ, jsou už z definice vágní a lidé o nich mají často různé představy. Díky své měkkosti jsou také hůře měřitelné, ale stejně jako racionalita jsou velmi důležité, dokonce i někdy důležitější, jelikož instinkty a emoce jsou zakořeněné hluboko v nás a také zrály po desítky tisíc let v průběhu evoluce lidstva. Z konstrukčního hlediska mozku je vrstva zodpovědná za emoce neblíže mozkovému kmeni a nervovému systému. Neuronové vzruchy z této vrstvy tedy musí urazit velmi krátkou vzdálenost pro přenos informace, reakce. Vzruchy z neokortexu, který je zodpovědný za racionální myšlení a

zdůvodnění, mají tuto vzdálenost o mnoho delší[34]. ***Lidé pracující v IT*** jsou z velké většiny ***univerzitně vzdělaní technici*** se znalostmi přírodních věd jako je matematika, fyzika či kybernetika (vyznávající měřitelnost, testovatelnost a dokazatelnost). Téměř nikdo však neabsolvoval v rámci svého vzdělání ***žádné přednášky s tématikou psychologie, sociologie, komunikace, motivace či vůdcovství***. Články z této oblasti mají pro nás technicky orientované pracovníky vždy jakýsi tajemný či alchymistický nádech. Proto budeme těmto hůře měřitelným měkkým aspektům věnovat stejnou pozornost, jako technickým znalostem a vzdělávání v technických dovednostech a budeme se snažit odkazovat na existující výzkumy.

Otevřená komunikace a spolupráce členů uvnitř týmu, stejně jako týmů (vývojového a údržby) se zákazníkem, je základ úspěšného projektu. Tato kapitola obsahuje nezbytné informace z oblasti psychologie a měkkých dovedností pro celkové pochopení významu týmové práce a měkkých dovedností. ***Smyslem této kapitoly je doplnění odpovědí na otázky v pozadí:***

- Proč mohou Agilní a Lean principy v praxi fungovat?
- Proč jsou výsledky těchto přístupů lepší, než u přístupu tradičního?
- Jaktože tradiční přístup není trvale udržitelný a generuje takové výsledky?
- Proč mi mí podřízení nerozumí, proč mne nenásledují?
- Proč ostatní slyší něco jiného, než já říkám?

Proto je dobré vědět také něco o psychologii týmu a jednotlivců, o jejich typologii, vědět a hlavně chápat, že ***lidé jsou různí, mají různý kontext, cíle a tím pádem i různou motivaci***, že jsme každý jiný a proč děláme určité věci, které ostatní nechápou, někdy ani my samotní ne. Motivací této kapitoly je uvědomění si, že vývoj softwaru není jen o naučených postupech, dovednostech (tzv. *hard skills*), ale také o měkkých vlastnostech (tzv. *soft skills*) jako je pochopení druhého, empatie, důraz na detaily či naopak celkový pohled. Tyto měkké dovednosti a vlastnosti jsou stejně kritické jako naučené znalosti a techniky, minimálně z pohledu fungování týmu, úspěchu projektu a naší spokojenosti. Pokud bude uvnitř týmu dusná atmosféra, určitě to nepřispěje ke kreativnímu řešení a očekávané produktivitě.

Ať už v práci či rodině, je pro komunikaci nejdůležitější znalost kontextu přijemce naší komunikace. ***Neznalost kontextu*** druhého člověka ***a domněnky způsobují většinu nedorozumění***. Přijdete například do práce unaveni po stěhování či rodinné hádce, chcete připravit chybějící podklady a vedoucí, který po vás chce standardní špičkový výkon, předpokládá, že jste podklady již připravili. Při diskusi s těmito domněnkami může dojít k zásadním rozporům. Stejně tak interpretace požadavků na software bez komunikace a vyjasňování s analytiky a zákazníkem může generovat nepoužitelnou funkčnost a těžce udržitelnou aplikaci či IT službu. Kontext je tedy velmi důležitým a vlastně nejdůležitejším bodem komunikace. Jelikož je zřejmé, že nikdy nemůžeme 100%ně znát kontext svých kolegů, kamarádů, rodinných příslušníků (pocity,

[34] To je také jeden z důvodů proč je tak těžké ovládat naše emoce. Díky funkci mozkového kmene *„bojuj nebo uteč“* jsme však přežily od pravěku až dodnes. Jelikož emoční vzruchy reagují dříve než jakákoliv myšlenka, musí jejich ovládnutí předcházet uvědomění si rozdílu mezi myšlenkou, emocí a naším já.

náladu, znalost či neznalost atd.) a musíme si být tohoto omezení vědomi a podle toho komunikaci vést. ***Druhým důvodem*** pro zařazení této kapitoly ***je nutnost chápat a respektovat odlišnosti, zájmy, cíle (a tudíž chování) druhých***. Tyto odlišnosti, zájmy, cíle a chování vyplývají z naší typologie, tělesné fyziologie (jak pracuje náš mozek) a také životního smyslu (cíle). Jejich popření, nevědomost, či kritika mohou vést k vnitřní nespokojenosti, která se projeví v komunikaci s ostatními a ve výsledcích práce jedince.

Snad jsme čtenáře namotivovali dostatečně. Pojďme tedy začít testem, který nám pomůže sestavit optimální tým vývojářů.

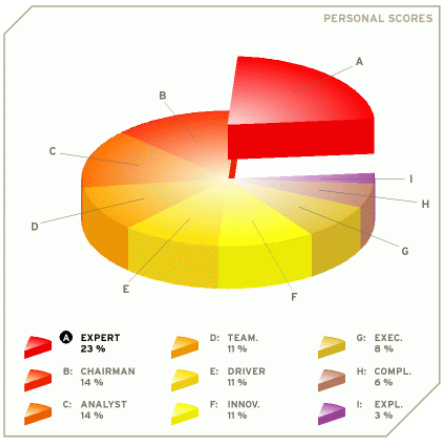
12.1 Týmové role - Belbin

Vývoj větších softwarových projektů je, stejně jako sport, týmová práce. Pokud tým není složen rovnoměrně (myšleno osobnostně), může to mít za následek zpoždění, či úplne zastavení vývoje. Někteří lidé mají schopnost přicházet s novými nápady, ale neumí je dotáhnout do konce. Jiní je naopak disciplinovaný a spolehlivý, ale je nepružný a může pomalu reagovat na nové možnosti. Již v 70. letech minulého století začal tento fenomén zkoumat psycholog Meredith Belbin a v roce 1981 vydal publikaci *Management Teams: Why They Succeed or Fail*, kde se poprvé objevilo rozdělení týmových rolí označované jako *The Belbin Team Inventory*. Pochopitelně tehdy nebyly informační technologie v takovém stav, aby se software vyvíjel v týmech. Původně byly role zaměřeny na tvorbu obecného týmu lidi tak, aby spolu lidé co nejlépe dosáhli výsledku v co nejkratším čase. Nástupem týmové práce ve vývoji software se ověřilo, že tyto principy lze s úspěchem zavést i na tvorbu programátorských týmů. Týmové role nejsou přímo ekvivalentem typologie osobností (jako např. MBTI). Týmové role vyjadřují, jak moc se testovaná osoba shoduje s jednou z devíti týmových rolí. Stejně jako u typologií osobností není člověk od přírody vyhraněn pouze pro jednu roli a můžou zde existovat překryvy. Cílem je v týmu pokrýt následujících 9 rolí rovnoměrně:

- **Inovátor** (Plant)
  - Přínosy: tvůrčí, dokáže řešit náročné problémy
  - Slabiny: velmi zaujatý vlastními myšlenkami na úkor komunikace
- **Výhledávač zdrojů** (Resource investigator)
  - Přínosy: rozvíjí kontakty, nadšený extrovert
  - Slabiny: je příliš optimistický, po počátečním nadšení většinou rychle ztratí zájem
- **Koordinátor** (Co-ordinator)
  - Přínosy: dává lidi dohromady, podporuje týmovou diskuzi
  - Slabiny: bývá manipulativní
- **Usměrňovač** (Shaper)
  - Přínosy: vyzývá k výkonu, je průbojný a odvážný
  - Slabiny: může urážet ostatní
- **Monitor vyhodnocovač** (Monitor Evaluator)
  - Přínosy: je stratég a má vysoké nároky
  - Slabiny: může chybět schopnost inspirovat ostatní
- **Týmový pracovník** (Teamworker)
  - Přínosy: je diplomatický, naslouchá

  - Slabiny: je nerozhodný v klíčových situacích
- **Realizátor** (Implementer)
  - Přínosy: má schopnost činit praktické kroky a akce
  - Slabiny: nepružný, pomalu reaguje na nové možnosti
- **Kompletovač finišer** (Completer Finisher)
  - Přínosy: pečlivý, svědomitý, plní termíny
  - Slabiny: neochotně nechává podílet se na své práci
- **Specialista** (Specialist)
  - Přínosy: poskytuje vědomosti a dovednosti, které jsou vzácné
  - Slabiny: přispívá pouze v úzké oblasti, zaobírá se převážně osobními zájmy

V základním rozdělení psychologických profilů (sangvinik, cholerik, melancholik, flegmatik) existuje jeden "správný" profil - sangvinik - a ostatní jsou svým způsobem diskriminační. V týmových rolích neexisuje nejlepší profil, Belbin říká, že pokud pokryjete tým všemi devíti rolemi, bude efentivně pracovat. Svou vlastní týmovou roli přiřazenou testem Belbin můžete zjisti zde: http://www.123test.com/team-roles-test/, výsledek pak může vypadat například takto:



Obr. 12-1: Prezentace výsledku testu týmových rolí

12.2 Typologie osobnosti MBTI

Zkratka MBTI znamená *Mayers-Briggs Type Indicator*. Jedná se o objektivní kategorizaci (poznání) osobností podle jejich preferencí (vrozených), tedy ne podle naučeného jednání. Nejedná se tedy o hodnotící metodu, neříká, jestli jsme špatní nebo dobří, modří nebo žlutí, ale je to opravdu jakýsi indikátor pomáhající nám pochopit, že každý jsme jiný. MBTI nám také pomáhá chápat zájmy a chování druhých, proč někdo slyší úplně něco jiného co říkám, proč musí mít přesnou hodnotu, když je přece jasné, že něco po sedmé mi jako informace stačí apod. Toto poznání není omezené jen na nějakou oblast, MBTI nám pomůže v zaměstnání, v práci, i při komunikaci s přáteli.

MBTI je založeno na poznání mentálních funkcí, nikoliv na náladách a dojmech (jako např. známý pojem melancholik) a důležitá je také zapracovaná dynamika. Lidé se v průběhu života mění, vyvíjí, jako dospívající máme jiné zkušenosti, potřeby a pocity, než ve stáří. MBTI vychází z poznání C. G. Junga

(4 mentální funkce jako základ) a doplňuje je o dynamický aspekt. MBTI je ve světě široce používaný nástroj pro poznání lidí (pozor, ne pro hodnocení!).

Následující tabulka ukazuje všech 16 osobnostních typů podle MBTI:

ISTJ	ISFJ	INFJ	INTJ
ISTP	ISFP	INFP	**INTP**
ESTP	ESFP	ENFP	ENTP
ESTJ	**ESFJ**	ENFJ	ENTJ

Barevně zvýrazněné typy budou sloužit dále v textu k vysvětlení opačných preferencí. Hned na úvod je nutné říct, že nejsme striktně jeden nebo druhý typ, striktně introvert či extrovert. Spíše tíhneme k jednomu z těchto pólů. Většinou uvidíme některé naše preference, vlastnosti v obou sloupečcích. Tíhneme tedy k tomu, kde máme většinu. Toto je důležité si uvědomit, aby se čtenář nedivil, až se najde v obou sloupcích.

12.2.1 Zaměření: introvert vs. extrovert

Prvním písmenem je naše zaměření, orientace, na myšlenky a pocity (introvert) či na slova, činy (extrovert). Následující charakteristiky značí introvertní orientaci:

- ponechává si názory pro sebe,
- vyhledává soukromí,
- koncentruje se na myšlenky,
- nejdříve myslí, pak jedná,
- sděluje až závěry,
- nerad je přerušován.

Extrovertní značí následující charakteristiky:

- vyjadřuje pohledy,
- vyhledává společnost,
- interakce s okolím,
- nejdříve jedná, pak myslí,
- přemýšlí nahlas,
- nečeká, až někdo domluví.

Pokud nás charakterizují obě oblasti půl na půl a přemýšlíme k čemu tíhneme spíše, pak nám pomůže následující klíč. Tím je, kde čerpáme energii, zda mezi lidmi nebo sami. Když jsme unaveni nebo máme špatnou náladu, sedneme si doma s knihou, pustíme si hudbu nebo vyrazíme sámi do přírody na kolo a relaxujeme (I). Když jsme unaveni nebo máme špatnou náladu, musíme mezi lidi, bavit se, zapomenout a načerpat energii (E).

12.2.2 Vnímání: intuice vs. smysly

Druhé písmenko S či N nám říká, jakým způsobem, zkrze co, vnímám své okolí a věci kolem nás. V zásadě platí, že intuitivní lidé mají něco jako 6.

smysl, tušení. Smysloví lidé vnímají smysly (čich, hmat, chuť) prožitkem. Intuitivní (N) lidé se vyznačují následujícími charakteristikami:

- vyhledává možnosti, nová spojení,
- celistvost, celkový obraz (les),
- orientuje se na budoucnost,
- zkoumá jak a proč věci fungují,
- vylepšuje řešení.

Smyslové (S) definuje spíše následující:

- Vyhledává fakta, přesné zadání, přesně definovaný výsledek,
- zaměření na detail (stromy, přes které mnohdy nevidíme les),
- užívá si přítomnosti,
- učí se, jak věci použít, ne proč fungují,
- používá ověřená řešení.

Následující nádherná ukázka rozhovoru smyslového (S) s intuitivem (N) vám asi pomůže nejlépe pochopit o čem je toto druhé písmenko.
S: *„Kolik je hodin?“*
N: *„Ještě je čas.“*
S: *„Ptám se kolik?!?”*
N: *„Je to v pohodě, stíháme.“*
S: *„Můžeš mi laskavě říct, kolik je hodin?!!!!“*
N: *„Sedm pryč.“*
S: *„A co to je 7 pryč!!!“*
N: *„Vždyť říkám, za 2min čtvrt na osm.“*
S: *„A tos mi to nemohl říct hned?“*

A pěkně začínající večer s romantickou večeří může dostat hned na úvod trhliny. Kdyby náš intuitiv věděl, že musí podat přesnou odpověď hned po optání, vyřešil by problém hned v zárodku.

12.2.3 Rozhodování: myšlení vs. cítění

Třetí písmenko T nebo F nám říká, jak se rozhodujeme, zda spíš podle toho, co je obecně pravdivé, platné (logika) či zda je to pro mě nebo pro ostatní důležité (emoce). Pro lidi myslící (T) platí následující:

- rozhodování na základě neosobní, objektivní logiky,
- pravda/nepravda,
- správně/špatně,
- osobně nezaujatý,
- cílem je poznání pravdy.

Lidé cítící (F):

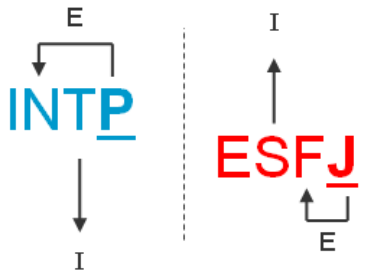
- se rozhodují se na základě subjektivních, osobních dojmů/hodnot,
- mám rád(a)/nemám rád(a),
- dobrý/špatný,
- vztahy a emoce,
- cílem je harmonie mezilidských vztahů.

Neznalost této preference u druhých osob může být opět velkým zdrojem problémů a nedorozumění. F nám připraví dobrou večeři a my T ani nepochválíme, jelikož jsme řekli na začátku vztahu, že umí výborně vařit a její jídlo milujeme a je přece jasné, že toto naše sdělení platí až do odvolání a nemusíme je jako T pořád opakovat, což však naše přítelkyně/manželka F nese velmi nelibě. T tedy může často působit chladně, i když uvnitř vroucně miluje a myslí si, že to dává najevo.

Opět krásná ukázka rozhovoru:
„A jak se má vaše malá dcerka?“ ptá se známý.
Matka F: *„No včera si ve školce hrála a představte si, co se jí stalo…“*
Otec T: *„Dobře.“*

12.2.4 Orientace funkcí: vnímání vs. usuzování

V případě vnímání (P) a rozhodování (J) se stejně jako v případě I/E nejedná o mentální funkci, ale o pomocný ukazatel, kterou z mentálních funkcí (tj. jestli vnímání nebo usuzování) používame při kontaktu s okolním světem. Toto je rozšíření MBTI systému původních Jungových tří rozměrů o čtvrtý rozměr. Tato vlastnost pomáhá jednoduše určit, zda procesy vnímání (S/N) a usuzování (T/F) budou obráceny dovnitř či ven a zda budou dominantní. Doplňující funkce pak bude orientována opačným směrem (viz následující obrázek).



Obr. 12-2: Orientace funkcí

Charakteristika usuzujících (J):

- systematičnost, schopnost důsledně se připravit na všechny možnosti,
- nutnost plánů a dlouhodobých představ, jak na věci jít a dělat je,
- trpí, pokud není rozhodnuto,
- potřebují rozhodnutí jako „mapu“,
- určují a dodržují termíny,
- začínají brzy, nemají rádi věci na poslední chvíli,
- „nejprve práce, potom zábava“,
- tendence organizovat život kolem sebe.

Kdežto vnímající (P) jsou charakterizováni následovně:

- uvolněnost, libování si v překvapeních, vést se proudem života,
- odkládají rozhodnutí, užívají si otevřených možností,
- rozhodují se pod tlakem (termínu),
- nesnášejí rutinu, vítají nové věci,
- „vyhrávají si“ s prací,

- spontaneita, tvořivost, flexibilita.

Opět je velmi jednoduché si představit možné třecí plochy lidí těchto odlišných typů. Lidé s orientací k usuzování (kterých se v populaci nadpoloviční většina) budou tíhnout k termínům, budou chtít konečná rozhodnutí, kdežto vnímajícímu P to bude připadat zbrklé, unáhlené, co když ještě může přijít jiná možnost.

12.2.5 Temperamenty

Preference probrané v předchozích kapitolách jsou pouze jednorozměrným pohledem bez hloubky. Jedinečnost každého typu nespočívá v prostém poskládání jednotlivých funkcí a orientací, ale v jejich vzájemném působení. Tato souhra také znamená, že určité chování nelze přičíst jediné preferenci, ale vždy pouze souladu všech. Nejužitečnější zkratkou k poznání jednotlivých charakteristik chování jsou tzv. temperamenty. Teorie temperamentů snad nejlépe vysvětluje a současně znázorňuje určité antropologické konstanty lidského chování [Ča05].

Strážce Idealista Racionál

Hráč	ISTJ	ISFJ	INFJ	INTJ
	ISTP	ISFP	INFP	INTP
	ESTP	ESFP	ENFP	ENTP
	ESTJ	ESFJ	ENFJ	ENTJ

Tabulka 12-1: Rozdílné temperamenty

Temperament ***hráče*** je prvním, o kterém si něco řekneme. Jádrem tohoto temperamentu je kombinace SP, smysly spojené s vnímáním. Jsou to umělci života, požitkáři, ctí řemeslo. Tito lidé jsou konkrétní v komunikaci, praktičtí v realizaci. Každý den přijímají s novou intenzitou, bez předsudků a svázanosti, žijí teď a tady, vyžadují vzruchy, neplánují (neumrtvují), nahrávají novým zážitkům. Chovají se podle výroku: pravidla jsou od toho, aby byla porušována. Hráči si vybírají povolání, která jsou bohatá na akce, zážitky, výzvy, často jsou hráči významnými umělci, herci, tanečníci, malíři (soustředení obrovského množství energie v daný okamžik), ale také hazardními hráči či reportážními fotografy. Literatura [Ča05] udává jejich zastoupení v populaci na 40 %.

Druhým temperamentem jsou ***strážci***. Strážci představují asi 38 % celkové populace [Ča05] a jejich jádrem je kombinace smyslů (S) a usuzování (J). Lidé s kombinací SJ toho tedy mají mnoho společného, ale mohou mezi nimi být i propastné rozdíly. Tento temperament je povolán k tomu, aby sloužil a byl užitečný, jedná se o strážce tradic, řádu, zákona, pořádku a systému. Tito lidé bývají konkrétní v komunikaci a kooperativní při realizaci cílů, jednou přijatá rozhodnutí nemění. Práci a život berou jako službu (povolání, nikoliv poslání). Mají respekt k formálním autoritám a funkcím. Pro řízení takového typu lidí je dobré vědět, že mají rádi hmatatelnou pochvalu.

Dalším temperamentem jsou ***racionálové*** tvořící kombinací intuice (N) a myšlení (T). Výskyt lidí s tímto temperamentem je vzácnější než u předchozích rolí, uvádí se přibližně 12 % [Ča05]. Lidé tohoto temperamentu žijí v trochu jiném prostředí, jakoby v jiném světě, obklopují je totiž převážně hráči a strážci. Ti tento pocit naopak nemívají, protože kolem sebe většinou najdou nějakou spřízněnou duši. NT charakterizuje touha věci zvládnout, proniknout k jejich podstatě. Skrze porozumění zákonitostem a schopnostem chtějí žít v souladu s nadčasovým řádem přírody. Jsou to architekti života, lačnící po poznání – dokonalosti (neustálá optimalizace všeho). NP jsou abstraktní v komunikaci, praktičtí v realizaci cílů - fakta, čísla, modely, principy. Pro svou práci vyžadují autonomii, autoritu uznávají jen na základě zásluh, nikoliv formálního postavení (primární skepse). Tento typ nejhůře dává a přijímá pochvalu. NP jsou povoláním často vědci, analytici, techničtí vizionáři (např. A. Einstein, B. Gates, Napoleon).

Posledním temperamentem, který jsme ještě nezmínili, jsou ***idealisté***. V tomto temperamentu se snoubí intuice (N) a cítění (F), proto vystihnout jeho povahu není lehké. Svět idealistů zůstává ostatním temperamentům trochu zahalen a naopak NF často nedokáží pochopit, za jakými přízemnostmi se ostatní pachtí.

Pro idealisty bývá nejdůležitější proces osobní proměny, proces stávání se něčím. K lepšímu pochopení je dobré říct, jak charakterizují tento typ Keirsey a Bates: NF jsou lidé, jejichž skutečné já spočívá v hledání vlastního já, pro něž smyslem života je mít smysl života. Jelikož jsou v neustálém procesu stávání se sama sebou, nikdy se sami sebou stát nemohou, protože dosažení cíle jej vlastně ruší a popírají (rozpor ve stávání se něčím a zároveň zůstat sám sebou). NF bývají abstraktní v komunikaci a kooperativní při realizaci cílů. Jsou charakterističtí také hledáním dalších významů a souvislostí – věcí mezi nebem a zemí. Jejich preferencí je morálka, ideál, čistota, vnitřní krása, integrita, opravdovost, přesvědčení. Často cítí, že mají jakési poslání (smyslem života je najít smysl života). NF bývají často přesvědčení vůdci vedoucí nás do rájů i pekel (např. Johanka z Arku, V.I. Lenin, M. Gándhí).

12.2.6 Test a další zdroje

Zájemci si mohou udělat na webu[35] jednoduchý test a zjistit, k jakému typu osobnosti inklinují. Na závěr zdůrazníme, že určitý typ podle MBTI ***nemá sloužit jako omluva našeho chování***, ale právě naopak, k lepšímu porozumění sebe sama a komunikaci nejen s kolegy, ale také s našimi blízkými, rodinou, kamarády. Navíc jednotlivé typy se navzájem potřebují: např. N udává směr a hledá nové cesty, rozjíždí byznys a S dotahuje věci do konce, dopracovává detaily. N nikdy sám do konce moc věcí nedotáhne, naopak S je rádo, že mu N řekne co a proč má udělat.

Ukázka rozdílného chápání odlišných temperamentů:
Při psaní těchto řádek si autor myslí, že zřetelně a jasně předává myšlenku a přitom se moc neopakuje. Toť pohled autora NTP. Čtenář intuitivní racionál s jistou zkušeností si bude při čtení opakovat: *„No jasně, vždyť je to zřejmé, že mě to nenapadlo (nebo takhle to přece děláme).“*

Naopak smyslový čtenář s podobnou praktickou zkušeností se může častěji zamýšlet a říkat: *„Hmm, více příkladů by pomohlo mému pochopení. Jak to myslí, co mám přesně dělat?“*

Proto je třeba vhodnou kombinací konkrétních příkladů a faktů, čísel a jemných návodů bez konečného řešení naplnit potřeby obou táborů, o což se také autoři tohoto textu snaží.

V rámci zjednodušení problematiky, jsme v této kapitole vypustili dynamiku funkcí, hierarchie funkcí (dominantní, vyvažovací, terciální a inferiorní), orientaci funkcí, které jsou pro pochopení klíčové. Pro zájemce doporučíme výbornou, česky psanou knihu od Michala Čakrta: *Typologie osobnosti* [Ča05], kde je daná problematika rozebrána podrobně.

[35] MBTI test: http://gardan.skeletus.com/TOOLS/Test/

12.3 Tým

Předchozí kapitola se zabývala jednotlivci a tím, proč se chováme tak, jak se chováme. Jak toto rozdílné chování zapadá do konceptu týmu a týmové práce? Týmová práce je jednou z nezbytných podmínek úspěšného provozu a údržby IT služeb. Různé osobnostní typy v týmu přináší jiný pohled. Intuitivní racionálové dodávají směr a vizi týmu či produktu, určují architekturu řešení. Smysloví pracovníci pak mají tendenci vidět i detaily, které mohou intuitivové přehlédnout a smysloví s důrazem na usuzování dotahují věci do konce, jsou tedy důležití pro úspěšné dokončení projektu. Když se řekne tým, většina lidí si představí skupinu spolupracujících či méně spolupracujících lidí. Je toto však úplná definice, která zahrnuje hlavní apekty týmu? Podívejme se na následující obrázek skupiny 3 lidí čekající na autobusové zastávce. Jedná se o tým nebo ne?

Obr. 12-3: Skupina: Skupina má rozdílné zájmy a cíle. Jeden člen skupiny by čekal hodiny, druhý nervozně vyhlíží autobus, třetí přemýšlí úplně nad jinými věcmi a autobus vůbec neřeší.

Následující příklad je mírně odlišný v jedné věci, která právě odlišuje skupinu od týmu.

Obr. 12-4: Tým: Tým je skupina lidí, která má společný cíl a zájem a všichni dělají co umí pro jeho naplnění.

Definice týmu podle [KaS] je následující:

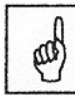
> *„A team is a small number of people with complementary skills who are committed to a common purpose, performance goals, and approach for which they are mutually accountable.“*
>
> *„Tým je menší skupina lidí se vzájemně se doplňujícími dovednostmi, kteří jsou oddáni společnému účelu, pracovním cílům a přístupu k práci, za něž jsou zvájemně odpovědni.“*

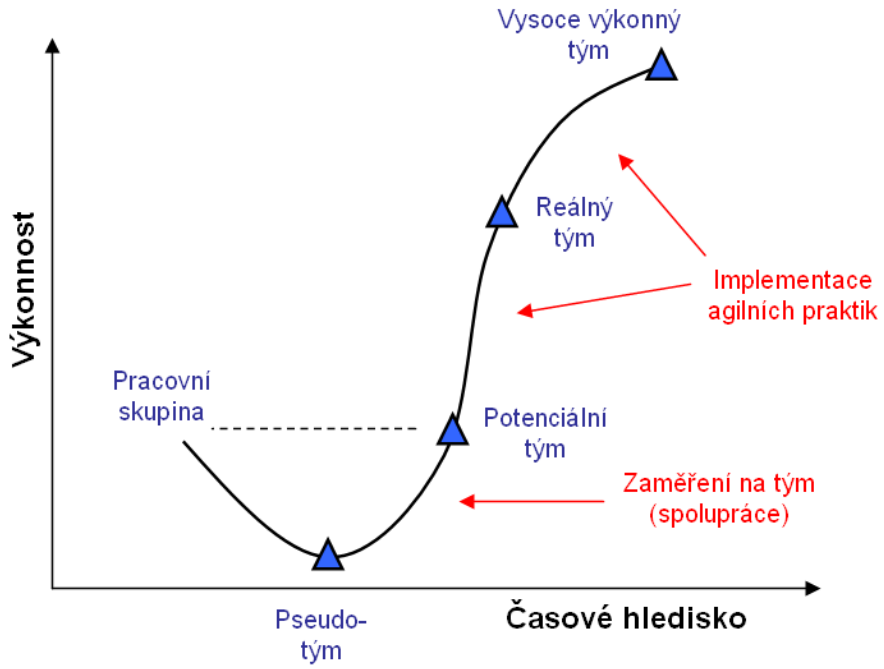
Klíčovým bodem, kde se tedy odlišuje tým od skupiny lidí, je oddanost společnému cíli a přístupu k práci. Bohužel, jako vždy je v praxi dosažení této ideální situace velmi obtížné, každý člen týmu má jinou motivaci a někdy bohužel i jiné cíle, podle definice se pak tedy nejedná o tým. Pokud se nám ale podaří dosáhnout společné oddanosti cílům, způsobu práce, je práce opravdu zábavou a to i ve chvílích stresu a pracovního vypětí, jak může sám autor potvrdit ze svých zkušeností.

Samozřejmě ne vždy je týmová práce aplikovatelná, například na manažerských postech musí často rozhodnutí přijmout (a také za něj nést hlavní odpovědnost) jedinec, ale to neznamená, že nepůjde pro podklady a názory nutné k rozhodnutí mezi ostatní manažery a mezi pracovníky a nebude s nimi diskutovat. Nejoblíbenější ***vládcové/manažeři byli ti, kteří sestoupili mezi lid a ptali se jich na jejich názory a potřeby***. Nedělali to samozřejmě pouze pro oblibu, která z tohoto přístupu pramenila, dělali to z toho důvodu, že daný vládce/manažer měl díky tomu kontext, věděl, co se opravdu děje v jeho království/firmě a jeho rozhodnutí potom byla také více relevantní a účinná.

Ze skupiny lidí, které dáme k sobě, se automaticky nestane tým. Lidé potřebují čas, aby se navzájem poznali, aby poznali silné stránky a slabiny ostatních a naučili se spolupracovat a vycházet spolu. Znalosti o vývoji týmu nám mohou pomoci v určitých stádiích jeho formování. Pokud o možných problémech víme, můžeme se snažit jim předejít či alespoň víme příčinu, pokud již problémy nastanou, což opět může pomoci problémy překonat. Již roku 1965 identifikoval Tuckman následující ***stádia formování týmu***: [Tu65]:

1. ***Formování*** – hledá se vedoucí týmu, ujasňuje se chování týmu, typicky je v této fázi mnoho klidu a harmonie.
2. ***Kvašení, burácení*** – začíná existovat konflikt, odpor proti úkolu, kontrole, způsobu práce. Tato fáze může trvat určitý čas a je důležitá pro úspěšné fungování týmu.
3. ***Vytváření norem*** – po určité době diskusí a vyjednávání se začínají utvářet normy a pravidla a lidé se učí společně pracovat, vzrůstá vzájemná podpora.
4. ***Fungování*** – jsou vyřešeny role v týmu, tým začíná efektivně fungovat.





Obr. 12-5: Efektivita týmu roste s jeho formováním. V tomto procesu mohou pomoci také agilní techniky a pravidla představené dříve.

Poté, co proběhlo formování týmu v prvních fázích, mohou v jeho dalším utváření a zvyšování efektivnosti pomoci právě agilní praktiky. Retrospektiva může pomoci týmu podívat se pravidelně na to, co nefunguje, rozebrat si proč a také přijmout konkrétní opatření. Párové programování umožní upevnit vazbu v týmu postupně, dvojice po dvojici. Denní mítinky umožní sdílet s členy týmu denní kroky, úspěchy a strasti a také umožní větší zaujetí členů týmu (všichni jsou informováni a měli by se zajímat).

Naopak největší chybou je po prodělání těchto fází a ukončení projektu tým rozpustit. V takové situaci se na každém novém projektu tým znovu formuje, poznává a efektivita práce roste zvolna a toto je také (kromě jiných) jednou z příčin zpoždění softwarových projektů. Naštěstí v kontextu provozu a údržby je tato situace lepší než v případě vývojových týmů. Výkonnost ale naopak ohrožuje vysoká fluktuace a učení se nových pracovníků. Zde však opět můžeme využít technik představených dříve a učení zefektivnit, hlavně cíleným a plánovaným použitím párové práce a učením se na reálných úkolech (jednoduché změny a incidenty).

Vzhledem k předchozímu je třeba zmínit ještě další fakt. Křivka pouze neroste, ale opět po určité době klesá dolů a kooperace členů týmu a jeho výkonnost upadá, protože chybí „nová krev“ a nové nápady. Samozřejmě záleží na lidech v týmu, práci, kultuře organizace, ale dá se říci, že nejpozději do dvou až tří let, kdy je tým spolu, je třeba přivést nový impulz a tým obměnit, rozpustit.

12.4 Koučing a mentoring

Tato kapitola se věnuje vysvětlení pojmů koučing a mentoring. V kapitolách o dovednostech, tvrdých praktikách, jsme zmiňovali důležitou roli mentora, a to

jak při zavádění nových praktik do práce týmu, tak při identifikaci problémů a jejich příčin. Zmiňovali jsme důležitost zkušené osoby, která pomůže s uvedením teoretických poznatků z knih do života (což je nejtěžším krokem adaptace nového způsobu práce), s rozpoznáním příčin a návrhem způsobu jejich odstranění. Toto je právě úloha osoby mentora. ***Mentoring*** je nezbytný pro úspěšný transfer znalostí a dovedností. Při implementaci nového procesu musí členové týmu změnit starý způsob práce, musí začít dělat věci jinak, v jiném pořadí, či se naučit úplně nové postupy. Mezitím však musí tým doručovat software minimálně stejně „dobře" jako předtím, musí být stejně efektivní jako doposud. Procesní mentor (agilní mentor, konzultant, ...) pomáhá právě v tomto: pomáhá život zjednodušit, vede členy týmu při implementaci nových postupů. Členové týmu potřebují znalosti a pomoc při změně postupů a tyto cíle pomáhá mentor naplňovat:

- Transfer znalostí – mentor sdílí s týmem zkušenosti z předchozích implementací praktik, procesů, postupů či agilních metod a podpůrných nástrojů. Důležitá je pravidelnost, pro každý tým bude toto sdílení trochu jiné (každý chápeme a učíme se jinak a každý projekt má specifická omezení a možnosti).
- Usnadnění změny.

Mentor by měl být přítomný v týmu po celou pracovní dobu, chodit mezi členy týmu a pomáhat, když potřebují, odpovídat na dotazy, vést workshopy. Dále také mentor povzbuzuje tým, ukazuje malé úspěchy, jelikož se jedná o změnu a každá změna je pro nás lidi těžká. Mentor nekritizuje ani nerozhoduje za tým a měl by následovat přístup: káži vodu, piji vodu (*practice what you preach*). Znamená to, že mentor jen nemluví, ale ukazuje prakticky v kontextu projektu jak danou věc provádět, vést, zachytit, modelovat.

Nejvhodnějším kandidátem na tuto roli je architekt, projektový manažer, vůdce v týmu, případně analytik se zkušeností z více projektů různých typů, který pomáhal se způsobem práce, s procesem (jeho úprava, tvorba, zlepšení, odstranění nadbytečných aktivit), má celkový rozhled a hlavně dobré analytické a komunikační dovednosti. Takový člověk by měl být schopný (ze zkušenosti a s využitím praktik a technik k tomu určených) vidět příčiny problémů a hlavně je umět vysvětlit. Skvělý analytik se znalostmi jak z byznys oblasti zákazníka, tak z technické oblasti, ale bez schopnosti vysvětlit příčiny a jejich důsledky je bohužel stejně přínosný jako člověk, který příčiny nevidí, či je nedokáže rozlišit od důsledků. Komunikační schopnosti jsou tedy nejdůležitější, jelikož denní náplní mentora je komunikace, vysvětlování a sdílení.

Pro objasnění konkrétních odpovědností a náplně práce mentora ukážeme příklad týdenního kalendáře a také vyjmenujeme možné oblasti záběru:

- Každodenní pomoc s implementací technik formou párové práce či formou workshopu.
- Vedení tréninků a příprava tréninkových materiálů v dané oblasti (agilní praktiky vývoje, provozu a údržby; ITIL, COBiT; podpora nástroji; definice a mapování metrik).
- Analýza současného stavu projektu s návrhem konkrétních kroků na zlepšení.

- Návrh způsobu práce, procesu, postupů ve spolupráci s členy týmu.
- Pomoc s automatizací jednotlivých kroků procesu nástroji (znalost nástrojů a jejich možností, nastavení).
- Pomoc s identifikací byznys cílů organizace a jejich propojením s ICT a ICT projekty.
- Definice metrik a způsob jejich sběru, měření.

Příklad týdenního kalendáře mentora:

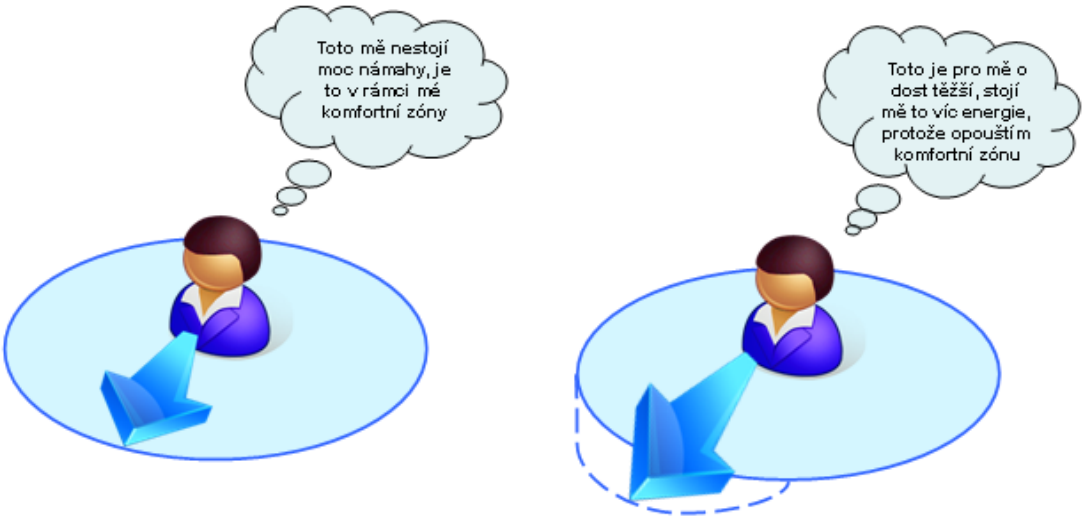
<table>
<tr><th>Hodina</th><th>Pondělí</th><th>Úterý</th><th>Středa</th><th>Čtvrtek</th><th>Pátek</th></tr>
<tr><td>8:00</td><td rowspan="2">Pravidelné plánování iterace w32 (tým SaM2)</td><td rowspan="2">Návrh operativních metrik tým SaM2</td><td rowspan="5">Principy agilního provozu a údržby (workshop s týmem ASM X1)</td><td>Angličtina A1</td><td rowspan="2">Blog pro firemní intranet</td></tr>
<tr><td>9:00</td><td rowspan="2">Rotace a párová práce na SPOC (tým SaM2)</td></tr>
<tr><td>10:00</td><td></td><td rowspan="2">Metriky: příprava prezentace (tým SaM2)</td><td>Retrospektiva (tým Cognos platforma)</td></tr>
<tr><td>11:00</td><td></td><td>Remedy demo pro tým MX3</td><td></td></tr>
<tr><td>12:00</td><td rowspan="2">Možnosti použití Remedy jako SPOC pro projekt MX3</td><td></td><td></td><td></td></tr>
<tr><td>13:00</td><td></td><td></td><td rowspan="2">Pravidelné hodnocení iterace w32 (tým SaM2)</td></tr>
<tr><td>14:00</td><td></td><td rowspan="3">Aktualizace tréninkových materiálů „Agilní provoz a údržba“ pro potřeby týmu ASM X1</td><td></td><td></td></tr>
<tr><td>15:00</td><td rowspan="2">Pravidlené sdílení znalostí: Agilní dokumentace (tým ASM X1, SaM2, MX3)</td><td></td><td rowspan="2">Příprava příspěvku na konferenci IEEE Provoz a údržba 2010</td></tr>
<tr><td>16:00</td><td></td><td></td></tr>
</table>

Druhým přístupem, který představíme v této kapitole je ***koučování***. Koučování staví na následujícím principu: místo toho, abychom se snažili my být co největší expert na své svěřence, vedeme je šikovnými otázkami (a nejen těmi) ke zvýšení jejich expertství. Jedná se o velmi starý princip posilující samostatnost, individualitu a tím také nezávislost jedince na sociální kontrole, což se ne vždy hodí či hodilo, ať již státnímu zřízení či různým organizacím [PP05]. Podle definice mezinárodní federace koučů[36] představuje koučing důvěryhodný vztah rozvoje, který napomáhá klientovi podniknout konkrétní kroky za účelem dosažení jeho vize, jeho cíle nebo přání. Koučink využívá procesů zkoumání a sebeobjevování k budování klientova uvědomění a přijetí zodpovědnosti, kterého dosahuje prostřednictvím větší struktury, podpory a aktivní zpětné vazby. Proces koučinku pomáhá klientovi nejen přesně definovat jeho cíle, ale také těchto cílů dosahovat rychleji a s větší efektivitou, než pokud by koučink nevyužíval.

Jedním z klíčů pochopení koučinku je mimo jiné jeho rozlišení několika druhů pomoci. Nás zajímá rozdělení na *pomoc* a *kontrolu* [PP05]. Spousta z nás asi zná ten pocit, že se někomu snažíme pomoc, ale ve skutečnosti je to vnucení,

[36] www.coachfederation.cz

naplnění naší představy (často je toto zřejmé při soužití či spolupráci se strážci – viz text výše), aniž bychom si to uvědomovali. Problémem pak je, že daný člověk není schopný bez této opakované naší „pomoci“ vyřešit danou věc sám, sám se o sebe postarat a my potom tvrdíme, že je neschopný a nevíme si s ním rady, stojí nás to hodně sil. Pokud se ale zaměříme na *pomáhání*, snahu daného člověka motivovat k tomu, aby sám hledal řešení a zkoušel různé možnosti, dostane se nám jako odměny jeho samostatnost a schopnost řešit zadané úkoly. Přesně ve smyslu starého čínského přísloví: *„Chyť člověku rybu a nasytíš ho na den, nauč ho jak ryby chytat a nasytíš ho na celý život.“*



Obr. 12-6: Opuštění komfortní zóny nás stojí energii, ale je jedinou cestou k růstu. Pravidelné opouštění komfortní zóny pomáhá zvyknout si na danou situaci, až se tato dostane do naší komfortní zóny a tudíž nás stojí míň energie. Právě v tomto pomáhá mentor/kouč. Kouč usměrňuje, otázkami navádí k překročení zóny, dodává odvahu k jejímu překročení, mentor dodává znalost a techniky nutné k překročení této zóny.

Kouč pomáhá člověku nezištně, snaží se motivovat. Předpokladem je, že každý má schopnosti se s daným problémem poprat, přičemž kouč danou znalost vůbec mít nemusí. Jedná se o tedy o pomáhání, ne řízení. Naopak mentor je odborník i v dané oblasti, dodává chybějící znalost, techniku, zkušenost. Dobrý mentor tedy musí být zároveň i koučem, jeho snahou je totiž být co nejdřív nepotřebný, což znamená osamostatnění se mentorovaného, mentorovaný sám je schopen si poradit a řešit úkoly, před kterými stojí.

Jednou z častých ***technik***, které kouč či mentor používá, je ***opakování otázek***. Po tom, co probere s partnerem daný problém, se kouč či mentor zeptá: *„Je to tedy jasné, co bys měl nyní udělat? Pokud ano, tak mi to zopakuj, shrň základní body.“*

Blíže se problematikou koučingu zabývá česky psaná kniha od autora Parmy [PP05], problematika mentoringu je zmíněna v anglicky psané knize [BR04].

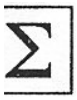
Kontrolní otázky:

1. Proč jsou tak důležité měkké aspekty při vývoji, provozu a údržbě softwaru?

2. Co je to MBTI?
3. Jaké charaktery podle MBTI znáte?
4. Jak se odlišuje tým od skupiny lidí?
5. Jaký je rozdíl mezi koučingem a mentoringem?

Úkoly k zamyšlení:
V této kapitole jsme si pověděli o důležitosti kontextu v komunikaci s druhými lidmi. Zamyslete se nad tím, jaké jsou možné cesty a způsoby zjištění kontextu druhých a jak jeho případná neznalost může komunikaci a pochopení komunikujících ovlivnit.

Korespondenční úkol:
V kapitole byl zmíněn odkaz na MBTI test, proveďte si tento test a na základě výsledku definujte možné třecí plochy vašeho typu s jinými. Výsledek zdůvodněte.



Shrnutí obsahu kapitoly
Kapitola pojednávala o měkkých aspektech vývoje softwaru, jeho provozu a údržby. Vysvětlili jsme důležitost znalosti kontextu v komunikaci. Zmínili jsme rozdíl mezi týmem a skupinou a cestu formování týmu. Náplní kapitoly bylo také představení typologie MBTI, která nám může pomoci lépe pochopit svoje potřeby a také potřeby druhých a na jejích základě upravit a zlepšit komunikaci s naším okolím. Součástí textu bylo také vysvětlení mentoringu a koučingu jejich rolí při osobním růstu, práci týmu a zavádění Agilních a Lean praktik.

13 Aplikace informačních systémů – Business Inteligence

V této kapitole se dozvíte:

- Co to je business inteligence a k čemu slouží.
- Jaké technologie a architektury v této oblasti používáme.

Po jejím prostudování byste měli být schopni:

- Porozumět umístění a smyslu dané oblasti v globální architektuře informačních systémů.
- Popsat základní architektury BI.

Klíčová slova této kapitoly:

ERP, BI, hvězda, sněhová vločka.

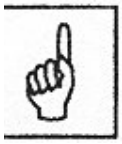
Doba potřebná ke studiu: 2 hodiny

Průvodce studiem

Kapitola představuje problematiku aplikace technologií a postupů pro získání manažerských dat z informačních systémů – tzv. Business inteligence.
Na studium této části si vyhraďte 2 hodiny.

V předchozím díle Informační systémy 1 se věnovala jedna z úvodních kapitol architekturám informačních systémů. Zmínili jsme se zde o možných variantách a aplikacích informačních systémů a o tom, na které vrstvě řízení podniku dané aplikace běží. Tyto aplikace, respektive části podnikového informačního systému, musí podporovat všechny tři úrovně řízení a rozhodování v organizaci (strategické, taktické, operativní), resp. musí k tomuto rozhodování poskytovat údaje, data a měly by také podporovat podnikové procesy na těchto úrovních. Následující obrázek ukazuje kontext, kdy jádrem podnikového IS je tzv. ERP systém, pro podporu komunikace se zákazníky slouží tzv. CRM systém, pro řízení dodavatelského řetezce pak tzv. SCM a v neposlední řadě na stategické a částečně i taktické úrovni tzv. BI systémy pro analýzy, přehledy a podporu rozhodování.

Obr. 13-1: Kontextový pohled na úrovně řízení a aplikace IS v podniku



Podnikové procesy, které existují v organizaci by měly být vhodně podpořeny informačními technologiemi, proto lze podle podpory různých podnikových procesů rozdělit aplikace informačního systému do několika základních kategorií, jedná se o:

- **ERP** (Enterprise Resource Planning) – jádro podnikového IS, zaměřené na řízení interních (převážně hlavních) podnikových procesů.
- **CRM** (Customer Relationship Management) – slouží pro podporu a automatizaci procesů směřovaných k zákazníkovi.
- **SCM** (Supply Chain Management) – slouží pro podporu a automatizaci dodavatelského řetězce.
- **BI** (Business Intelligence), často součást manažerských IS (tzv. MIS), sbírá data z předchozích a vytváří agregace, analýzy, trendy, které slouží pro podporu rozhodování manažerů.

Dále v textu se budeme podrobněji zabývat právě Business Inteligence, jelikož je to jeden z trendů dnešních aplikací informačních systémů.

Co si tedy představit pod pojmem Business Intelligence? Podle [No05] je termín BI použit pro označení celého komplexu činností, úloh, přístupů a technologií, které téměř výlučně podporují analytické a plánovací činnosti podniků a organizací (a nyní jsou běžnou součást informačních systémů) a jsou postaveny na principu multidimenzionality = pohled na realitu z více možných úhlů.

Česká společnost pro systémovou integraci (ČSSI[37]) definuje business intelligence jako sadu procesů, aplikací a technologií, jejichž cílem je účinně a účelně podporovat rozhodovací procesy ve firmě.

Aplikace BI pokrývají analytické a plánovací funkce většiny oblastí podnikového řízení (tj. prodej a nákup, skladování, marketing, finanční řízení, výroba, správa lidských zdrojů apod.). Do nástrojů BI tedy zahrnujeme:

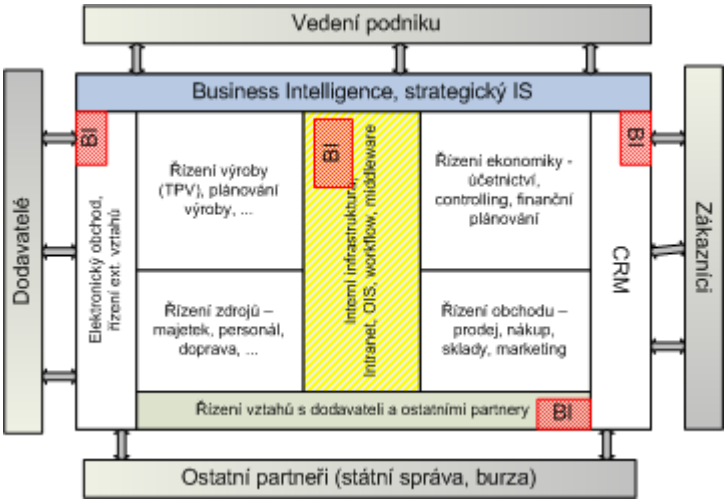
- produkční (provozované), zdrojové systémy,
- dočasná datová úložiště (DSA – Data Staging Area),
- operativní uložení dat (ODS – Operational Data Storage),
- transformační nástroje (ETL – Extraction Transformation Loading),
- integrační nástroje (EAI – Enterprise Application Integration),
- datové sklady (DWH – Data Warehouse)
- datová tržiště (DMA – Data Marts)
- OLAP,
- reporting,
- manažerské aplikace (EIS – Executive Information System),
- dolování dat (Data Mining),
- nástroje pro správu meta-dat
- apod.

Cílem BI aplikací není pouze shromažďovat data, ale hlavně podporovat manažerská rozhodnutí (na základě poskytnutých dat). BI systémy poskytují

[37] www.cssi.cz

historická a současná data stejně jako předpovědi, jak se může vyvíjet situace na trhu či jaká jsou rizika, a to hned v několika scénářích v návaznosti na manažerská rozhodnutí.

Obr. 13-1 nám ukázal místo BI v řídící globální architektuře podniku. Pokud se ponoříme více do detailů, zjistíme, že BI je velmi provázáno s ostatními aplikacemi IS, jelikož z nich čerpá data a často také jiná do těchto systémů vrací. Postavení BI v aplikační architektuře IS ukazuje následující obrázek.



Obr. 13-2: Postavení BI v aplikační architektuře IS

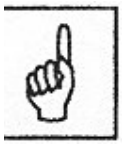
13.1 Základní principy řešení BI

Informační systémy pracují se dvěma základními typy informací – operativními a analytickými. Operativní informace slouží pro realizaci podnikových transakcí. Jsou uloženy většinou v relačních databázích a zobrazují aktuální stav podniku, v průběhu jednoho dne se mohou i několikrát měnit (např. účetnictví, objednávky). Transakční systémy realizují jejich zpracování v reálném čase a označují se jako **OLTP** (On Line Transaction Processing) systémy. Vzhledem k analytickým aplikacím se data OLTP chápou jako primární, zdrojová data. Systémy pracující s analytickými informacemi využívají tato primární data vytvořená v OLTP systémech. V 80. létech minulého století se pro tyto systémy vžil pojem **OLAP** (On Line Analytical Processing), ale ten se nyní v podstatě překrývá s pojmem BI. OLAP systémy jsou tedy míněny v užším významu, spíše jako technologie, která je charakterizovaná:

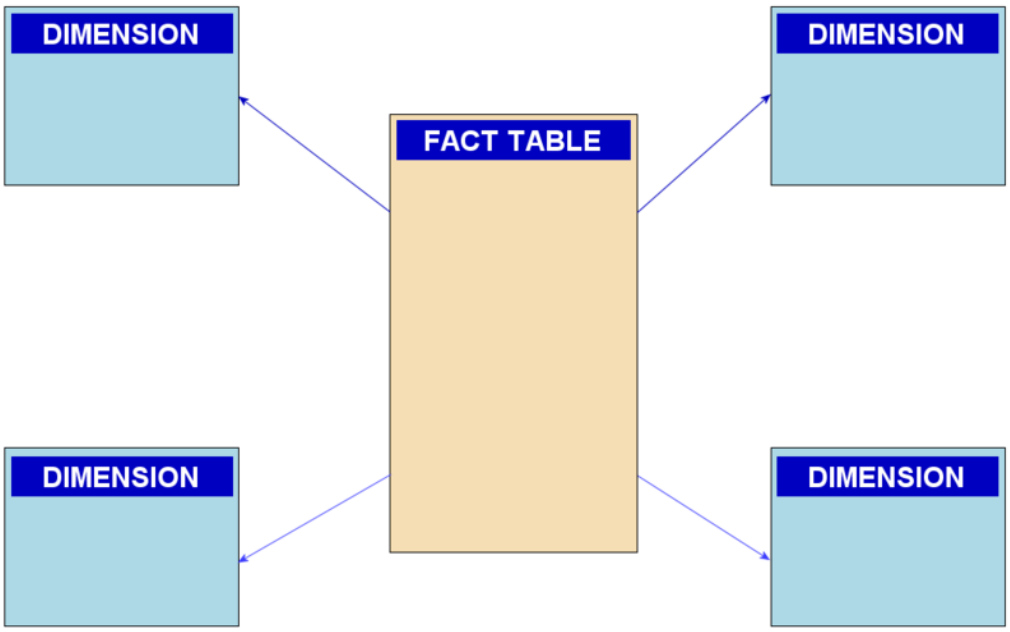
- poskytováním informací na základě primárních dat (z OLTP systémů),
- použitím multidimenzionálních databází,
- použitím agregací dat,
- zachycením časového faktoru (umožňují tedy časové srovnání, predikovat trendy).

V předchozím výčtu byl zmíněn pojem **multidimenzionální databáze**. O co se jedná? Jak může vypadat její struktura v relační databázi? Začněmě nejdříve vysvětlením, o co se jedná. Pro transakční systémy je typické, že data jsou uložena v 3. normální formě, což se ale pro data analytického typu nehodí. Pro

poskytování analýz a různých pohledů je nutné, abychom se mohli na data dívat z více hledisek současně (tzv. multidimenzionální pohledy). To je ale pro data v 3. NF velký problém. Analytické nástroje koncového uživatele musí umožňovat nacházení souvislostí, které nejsou z primárních dat na první pohled zřejmé. Navíc procházíme velké množství dat, je nutné počítat agregace (v DB v 3.NF nejsou uloženy) a ukládat je také do přehledných tabulek a grafů. Multidimenzionální databáze jsou již optimalizovány pro uložení a interaktivní využívání multidimenzionálních dat. Výhodou těchto OLAP technologií je rychlost zpracování a efektivních analýz dat. Základní princip BI aplikací je tedy multidimenzionální tabulka umožňující pružně měnit jednotlivé dimenze a nabízet tak různé pohledy.



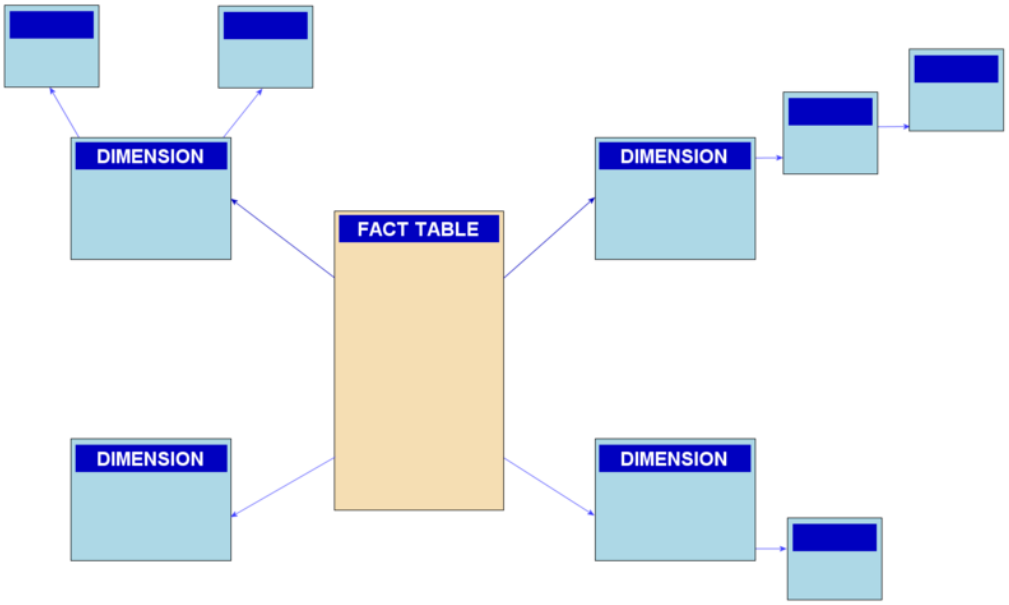
Datové modely produkčních systémů jsou velmi složité, obsahují mnoho tabulek a vazeb, často proprietárních optimalizovaných „normálních forem“ či částečně objektových, resp. XML dat. Pro dotazy přes více tabulek je nutné vytvářet propojovací můstky a složité dotazy a ty jsou pak často největší zátěží systému. Proto se objevilo zjednodušení ERD modelu databáze. Jedná se o přizpůsobení struktury pro tvorbu datových skladů a přiblížení dat koncovému uživateli. Tímto vznikl relační **dimenzionální model**, často také nazývaný „schéma hvězdy“, resp. schéma sněhové vločky (viz Obr. 13-3 a Obr. 13-4).



Obr. 13-3: Star schéma – schéma hvězdy

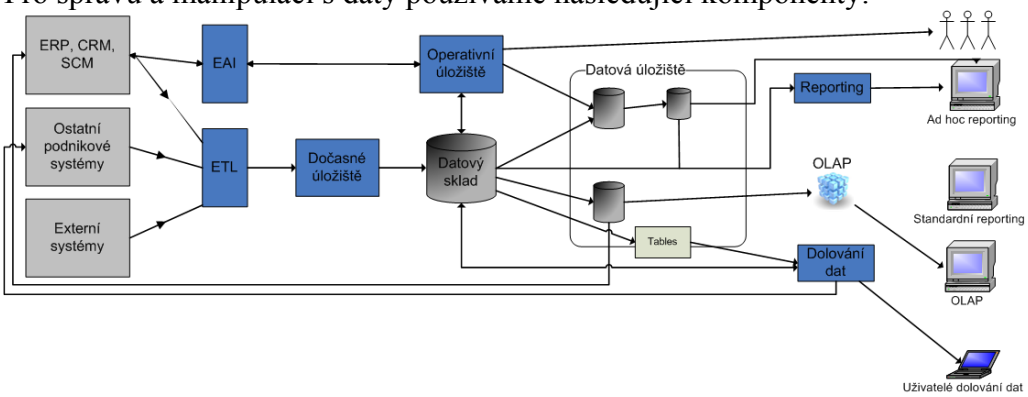
V centru schématu je tabulka sledovaných faktů, tj. například potřebné ekonomické ukazatele, které jsou identifikované klíčem složeným z klíčů tzv. multidimenzionálních tabulek, v nichž jsou uloženy prvky jednotlivých dimenzí (v podstatě číselníky dimenzí). V praxi někdy bývá problém rozhodnout, zda pole zařadit do tabulky faktů či do tabulky dimenzí. Pomůckou je následující:

- Tabulka faktů – sledovaná veličina je měřitelná a měnící se v čase.
- Tabulka dimenzí – sledovaná veličina je diskrétní a vystupuje spíše jako konstanta.



Obr. 13-4: Snow flake schéma

Pro správu a manipulaci s daty používáme následující komponenty:



Obr. 13-5: Komponenty BI

Pojďme si nyní stručně jednotlivé komponenty z obrázku popsat:

- **Produkční (zdrojové systémy)** – systémy, z nichž BI aplikace získávají data a nepatří do skupiny BI aplikací. Tyto systémy podporují modifikaci a ukládání dat v reálném čase, nejsou ale navženy pro analytické úlohy.
- **ETL (Extraction, Transformation and Loading) systémy** – datová pumpa, jedna z nejvýznamnějších komponent BI. Úkolem ETL je z datových zdrojů získat a vybrat data, upravit do požadované formy, vyčistit a nahrát do specifických struktur datového skladu. ETL pracují v dávkovém režimu (batches), dávky mohou mít denní, týdenní, měsíční intervaly.
- **EAI (Enterprise Application Integration) –** převážně používány ve vrstvě zdrojových systémů, jejich cílem je integrace primárních podnikových systémů a redukce vzájemných rozhraní (integrace dat či funkcí). EAI platforma pracuje v reálném čase, ne v dávkách jako ETL. Své využití v BI mají hlavně jako vrstva datové integrace.

- **Dočasné úložiště dat (DSA) –** komponenta slouží pro dočasné ukládání extrahovaných dat z produkčních systémů. Cílem komponenty je podpořit rychlou a efektivní extrakci dat. Jedná se tedy o první komponentu BI řešení (nepovinnou), sloužící pro ukládání netransformovaných zdrojových dat. Uplatnění má především u neustále zatížených systémů (nezatěžovat ještě více daný systém) a systémů, kde je třeba data konvertovat do DB formátu (např XML či textová data).
- **Operativní úložiště dat (ODS) –** opět nepovinná komponenta BI řešení. Existují dvě definice ODS, jedna říká, že slouží jako jednotné místo datové integrace aktuálních dat z primárních systémů a podporují vkládání a modifikaci dat v reálném čase. Typicky jsou napojené na EAI platformy. Druhá definice vymezuje ODS jako databázi navrženou s cílem podporovat relativně jednoduché dotazy nad malým množstvím aktuálních analytických dat. Zásadní rozdíl mezi DSA a ODS je v použití. DSA je pouze dočasné úložiště dat před jejich zpracováním v datovém skladu (pak jsou data vymazána). ODS oproti tomu slouží jako databáze podporující analytický proces.
- **Datový sklad (DWH) –** v dnešní době jeden z nejvýznamnějších trendů v rozvoji podnikových informačních systémů. Datový sklad je definován[38] jako integrovaný, subjektově orientovaný (data rozdělena podle jejich typu), stálý (read-only, data jsou čerpána z operativních databází) a časově rozlišený souhr dat uspořádaný pro podporu potřeb managementu.
- **Datová tržiště (DMA) –** princip je podobný jako u datových skladů, jen s tou výjimkou, že DMA slouží omezenému okruhu uživatelů (oddělení). Může to být mezistupeň k datovému skladu.
- **OLAP databáze –** jedná se o jednu či více souvisejících OLAP kostek, které většinou obsahují (na rozdíl od datových skladů) předzpracované agregace dat podle definovaných hierarchických struktur dimenzí a jejich kombinací.
- **Reporting –** činnosti spojené s dotazováním se do databází pomocí standardních rozhraní těchto databází (typicky např. SQL u relačních DB). Definujeme standardní reporting, kdy v určitých časových periodách spouštíme předpřipravené dotazy a ad hoc reporting, kdy jsou databázi dotazovány uživatelem vytvořenými explicitními dotazy.
- **Dolování dat (data mining)** – umožňuje pomocí speciálních algoritmů automaticky objevovat v datech strategické zavislosti. Tato analytická technika je pevně spjata s datovými sklady. Důležitou vlastností dolování dat je, že se jedná o analýzy (prediktivních informací) odvozené z obsahu dat, nikoliv předem specifikované uživatelem/vývojářem. Dolování dat slouží k odhalování nových skutečností, což pomáhá zaměřit pozornost na podstatné faktory podnikání. Matematické a statistické techniky využívané k dolování dat jsou například rozhodovací stromy, neuronové sítě, genetické algoritmy, clustering a klasifikace.
- **Manažerské aplikace (EIS)** – jedná se o typ aplikací integrující v sobě všechny nejdůležitější datové zdroje systému významné pro řízení

[38] Podle zakladatele Datových skladů Billa Inmona.

organizace jako celku. S tím jsou spojené i specifické nároky na prezentaci aplikací a jejich zpřístupnění vedoucím pracovníkům firmy. EIS podporuje tedy manažerské procesy (podnikové analýzy, plánování, rozhodování), na rozdíl od reportingu spíše vyšší a střední úroveň managementu.

13.2 Aplikace BI

Aplikace BI lze v podstatě využít ve všech oblastech lidské činnosti, kde sledujeme a vyhodnocujeme hodnoty určitých ukazatelů. Díky tomute existuje spousta BI řešení pro určité oblasti s patrným překryvem. Následující text stručně představuje oblasti použití BI aplikací.

13.2.1 Finance

BI aplikace v této oblasti umožňují získat hodnoty ukazatelů finanční výkonnosti za jednotlivé projekty, závody či za celou organizaci. Výstupem mohou být panely obsahující možnost zjištění odchylek od plánovaných hodnot a okamžitě zjistit místo, kde dochází k problémům a přijmout odpovídající opatření. Nasazení BI aplikací v oblasti finančnictví s sebou přináší vysokou transparentnost financní a řízení nákladů (velmi výhodné například pro sledování projektů).

BI finanční aplikace jsou používány převážně v následujících oblastech:

- Finanční plánování a prognózy (hlavně analytické nástroje).
- Finanční výkaznictví a konsolidace (rychlá tvorba složitých výstupů).
- Analýzy nákladů a ziskovostí (skutečné náklady spojené s produkty, partnery zákazníky).
- Řízení rizik (sledování rizik spojených s finančními operacemi a přijímání akcí).

13.2.2 Výroba

Také jedna s klíčových domén BI aplikací. Přehledy o historii a aktuálním stavu výroby, kontrola jakosti jsou nezbytné části efektivního řízení výrobního procesu. Aplikace BI ve výrobní sféře jde tedy nalézt především následující:

- Plánování a monitorování klíčových ukazatelů výrobního procesu (dodávky v porovnání s plánem, doba trvání výrobního cyklu, rozpracovaná výroba, obrat zásob, kvalita) přes závody, produkty, materiál.
- Analýzy a plánování trendů založené na historických datech – speciální aplikace BI určené k plánování a simulaci výrobního procesu na základě historických dat.
- Podpora nástrojů automatizovaného řízení výrobního procesu – dodání informací pro nápravu odchýlených ukazatelů od plánu v případě automatizovaného řízení výrobního procesu.

13.2.3 Logistika

Poslední ukázkou použití BI aplikací je oblast logistiky. Informace o průběhu vyřizování objednávek umožňují sledovat efektivnost celého procesu dodávky

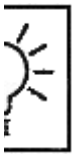
zboží, tak i pouze jednotlivých částí. Díky automatizaci ICT/IS mají BI aplikaci k dispozici spoustu informací a je možné je použít v širokém spektru služeb:

- Analýza efektivnosti dopravců – analýza chování externích (najmutých) dopravců, např. dodržení plánovaného termínu dodávky, náklady, splnění požadavků na přepravu. Tyto analýzy mohou být prováděny přes jednotlivé dopravce, typy zboží, geografickou oblast.
- Analýza dopravních nákladů spojených s různými druhy dopravy, dodacími podmínkami, uzavřenými rámcovými smlouvami (umožní vybrat nejefektivnější pro konkrétní zboží).
- Kapacitní plánování (mohou dodávat potřebné informace plánovacím softwarům).
- Analýza doby dodávky.
- Analýza důvodu problémů a reklamací.

Pro zájemce o danou problematiku odkážeme na českou literaturu [No05] či aplikačně zaměřenou [So06].

Kontrolní otázky:

1. Co je to BI?
2. Jaké znáte komponenty BI?
3. Jaké využití a v jakých oblastech má podle vás BI?



Úkoly k zamyšlení:

Zamyslete se nad využitím BI aplikací ve výrobním procesu. Dokážete definovat některé ukazatele, které jsou důležité pro analýzy a plánování trendů založené na historických datech? Zodpovězte si také, proč právě tyto ukazatele.

Korespondenční úkol:

V části hovořící o OLAP databázích jsme nezmínili základní druhy OLAP databází: MOLAP, ROLAP, DOLAP. Najděte v literatuře či na Internetu a stručně popište, k čemu dané typy OLAP slouží a zdůvodněte, proč je asi odlišujeme (ne více než 2 strany textu, vyjma obrázků).

Shrnutí obsahu kapitoly

V této kapitole jsme představili problematiku BI, její místo v podnikové infrastruktuře. Dále jsme stručně představili jednotlivé komponenty BI včetně použití BI aplikací.

14 Literatura

[AgM] Manifest agilního vývoje. Dostupné na: [http://www.agilemanifesto.org/].

[AP1] Brown, J., W. et al.: *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons. 1998. ISBN 0-471-19713-0.

[Be99] Beck, K.: *Extreme Programming Explained. Embrace Change*. Addison-Wesley. 1999.

[BR04] Bergstrom, S., Raberg, L.: *Adopting the Rational Unified Process: Success with the RUP*. Addison-Wesley 2004. ISBN 0321202945.

[BF75] Brooks, P., F.: *Mythical Man-Month: Essays on Software Engineering. Anniversary Edition*. Addison-Wesley. 1995 (originally printed 1975). ISBN 0201835959.

[Bu04] Buchalcevová, A.: *Metodiky vývoje a údržby informačních systémů*. Grada. 2005.

[Co04] Cockburn, A.: *Crystal Clear. A Human Powered Methodology for Small Teams*. Addison-Wesley. 2004. ISBN 0201699478.

[Con95]Constantine, L. L. (1995) *Constantine on Peopleware*. Englwood Cliffs, NJ, Yourdon Press.

[Cr08] Crisp web: Planning poker. Dostupné na: [old.crisp.se/planningpoker/].

[Ča05] Čakrt, M.: *Typologie osobnosti. Přátelé, milenci, manželé, dospělí a děti*. Praha. Management Press. 2005. ISBN 80-7261-112-7.

[dS05] de Souza, S. C., Anquetil, N., de Oliveira, K. M. (2005): A study of the documentation essentials to software maintenance. *In: SIGDOC 2005*, Coventry, UK, pp. 68-75

[dS07] de Souza, S. C., Anquetil, N., de Oliveira, K. M. (2007): Which documentation for software maintenance? *Journal of the Brazilian Computer Society*. 13(2), 31-44

[Do81] Doran, George T.: *There's a S.M.A.R.T. way to write managements's goals and objectives*. and Miller, Arthur F. & Cunningham, James A "How to avoid costly job mismatches" Management Review, Nov 1981, Volume 70 Issue 11

[EUP] Ambler, S. W., Nalbone, J., Vizdos, M. J.: *Enterprise Unified Process: Extending the Rational Unified Process*. 1. Edition, Prentice Hall, 2005

[Ga08] Gartner: *SaaS Adoption Trends in the U.S. and U.K.* [online, 8.7.2009].

Získáno 25.7.2009. URL: <http://www.gartner.com/it/page.jsp?id=1062512>

[Ga09] Gartner: *Gartner's Hype Cycle Special Report for 2009*. [online, 31.7.2009]. Získáno 20.8.2009. URL: <http://www.gartner.com/DisplayDocument?id=1108412>

[Gr] Grässle, P., et al: *UML 2.0 in action – project based tutorial*. Galileo Press. 2004. ISBN 1-904811-55-8

[Ho08] Hopkins, R., Jenkins, K.: *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. IBM Press. 2008

[IEEE] IEEE Std. 610.12 Standard Glossary of Software Engineering Terminology, IEEE, Computer Society Press, Los Alamitos, CA, 1990.

[IEEE2]IEEE 1219 Standard for Software Maintenance, www.iso.org

[IBM1] IBM: *UML Basics : Class diagram*. [online] (cited 2012-07-26) URL<http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/>

[IBM2] IBM: *UML basics: An introduction to the Unified Modeling Language*. [online] (cited 2012-07-26) URL<http://www.ibm.com/developerworks/rational/library/769.html>

[IBM3] IBM: *UML basics: The sequence diagram*. [online] (cited 2012-07-26) URLhttp://www.ibm.com/developerworks/rational/library/3101.html

[IBM4] IBM: *UML basics: The component diagram*. [online] (cited 2012-07-26) URL<http://www.ibm.com/developerworks/rational/library/dec04/bell/index.html>

[ISO] ISO 12207:2008 Standard for Software Lifecycle Processes, http://www.iso.org

[ISO1] ISO 14764:2006 Standard for Software Lifecycle Processes – Maintenance, www.iso.org

[ISO2] ISO 20000 Standard for Service Management: Part 1, Part 2, www.iso.org

[ITIL] OGC ITIL version2, version3. URL: <http://www.ogc.gov.uk>

[ITCZ] itSMF Czech Republic: *Úvodní přehled ITIL V3*. 2007. URL: <http://www.itsmf.cz/>

[JSE] Java SE homepage: http://java.sun.com/javase/

[JEE] Java EE homepage: http://java.sun.com/javaee/

[JME] Java ME homepage: http://java.sun.com/javame/

[Kr03a] Kroll, P., Kruchten, P.: *The Rational Unified Process. Made Easy*. Addison-Wesley. 2003. ISBN 0321166094.

[Kr03b] Kroll, P., Kruchten, P.: *The Rational Unified Process. An Introduction*. Addison-Wesley. 3. vydání. 2003. ISBN 0321197704.

[Kr05] Kroll, P., Royce, W.: *Key principles for business-driven development*. Dostupné na Rational Edge: [www.ibm.com/developerworks/rational/library/oct05/kroll/index.html?S_TACT=105AGX15&S_CMP=EDU].

[KaS] Katzenbach, J., R., Smith, D., K.: *The Wisdom of Teams: Creating the High-performance Organization.* Boston: Harvard Business School. 1993

[Lu05] Luca, D., J.: *Feature Driven Development Overview Presentation*. 2005. Dostupné na: [www.nebulon.com/articles/fdd/download/fddoverview.pdf].

[Ma02] Mahmoud, Q., H.: *Naučte se Java 2 Micro Edition*. Grada. 2002.

[Ma99] Polo, M.; Piattini, M.; Ruiz, F.; Calero, C.: *MANTEMA: a software maintenance methodology based on the ISO/IEC12207 standard.* In: Fourth IEEE International Symposium and Forum on Software Engineering Standards 1999, pp. 76 – 81.

[MAN] Polo, M., Piattini, M., Ruiz, F.: *Advances in Software Maintenance Management: Technologies and Solutions*. IGI Publishing. 2002.

[Mat] Materna Information & Communications: *Průzkum stavu řízení IT v zemích Evropy*. ČSSI Praha. Systémová integrace. roč. 16, č. 1, str. 98-105. Duben 2009

[Mm06] Melnik, G., Maurer, F.: *Comparative Analysis of Job Satisfaction in Agile and Non-agile Software development Teams*. In. Proceedings of the 7th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2006). LNCS 4044, pp. 32-42

[Mm07] Melnik, G., Maurer, F.: *Job Satisfaction and Motivation in a Large Agile Team*. In. Proceedings of the 8th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2007). LNCS 4536, pp. 54-61

[MS1] Microsoft Operatios Framework, verze 4.0 (2009), http://technet.microsoft.com/en-us/library/cc506049.aspx .

[No05] Nový, O., Pour, J., Slánský, D.: *Business Intelligence. Jak využít bohatství ve vašich datech*. Grada. 2005. ISBN 80-247-1094-3.

[Nos92] Nosek, J. T. (1998) The Case for Collaborative Programming. *Communications of the ACM*. Březen 1998, pp. 105-108

[Op07] Opelt, K. (2007) Overcoming Brook's Law. *In: Proceedings of 8th International Conference XP 2007*, Berlin: Springer, pp. 175-178

[OUP] OpenUP homepage. Dostupné na: [www.eclipse.org/epf/general/getting_started.php].

[Pa1] F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, M. Stal: *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons. Chichester. 1996. Dostupné na: [http://hillside.net/patterns/books/Siemens/abstracts.html].

[Pa2] Avgeriou, P., Zdun, U.: *Architectural Patterns Revisited – A Pattern Language*. Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications. Pp. 133-146. ACM SIGPLAN 2005. ISBN1-59593-031-0.

[Pe] Pender, Tom: *UML Bible.* John Wiley & sons. 2003. ISBN 0-7452-60-49.

[PP05] Parma, P.: *Umění koučovat*. Alfa Publishing. Praha 2006. ISBN 80-86851-34-6.

[Po06] Poppendieck, M., Poppendieck, T.: *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley Professional. Boston. 2006.

[Po09a]Poppendieck, M.: *Deliberate Practice in Software Development*. Příspěvek na konferenci Agile 2009.

[PG06] Pollice, G.: *Process adoption anti-patterns: How -- not -- to make a process work for you*. IBM developerworks. Dostupné na: [www.ibm.com/developerworks/rational/library/dec06/pollice/index.html].

[RE] Rational Edge homepage (RUP články): [http://www-128.ibm.com/developerworks/views/rational/libraryview.jsp?topic_by=rup%20(rational%20unified%20process)].

[Sa09] Rozhovor s Michaelem Stallardem: *Fired up or burned out?* http://www.ibm.com/developerworks/podcast/dwi/cm-int060909.mp3

[SEI1] Bass, L., Clements, P., Kazman, R.: *Software Architectures in Practice*. Addison-Wesley. 2003. ISBN 0-321-15495-9.

[SG] Standish Group (2009): Chaos research 1999-2009.

[Sch04] Schwaber, K.: *Agile Project Management with Scrum.* Microsoft Press 2004.

[Sp02] Spell, B.: *Java. Programujeme profesionálně*. Computer Press. 2002.

[So06] Sodomka, P.: *Informační systémy v podnikové praxi*. Computer press. Brno. 2006. ISBN 80-251-1200-4.

[Sr] Sriganesh, Brose, Silverman: *Mastering Enterprise JavaBeans 3.0*. Dostupné na: http://www.theserverside.com/tt/books/wiley/masteringEJB3/index.tss

[Tu65] Tuckman, B., W.: Developmental Sequence in Small Groups. *Psychological Bulletin*, Vol. 63, pp. 384-399. 1965

[Wi05] Wissel.net: *Crystal Mind map*. Posted 2005. Dostupné na: [www.wissel.net/blog/d6plinks/SHWL-6DVDFT].

[We97] Wegner, P.: Why Interaction is more powerful than algorithms, *Communications of ACM*, 40(5):80-91, 1997

[WK99] Williams, L. A., Kessler, R. R. (1999) All I Really Need to Know about Pair Programming I Learned in Kindergarted. *Communications of the ACM*, vol. 43, issue 5, pp. 109-114